A decentralized analysis of multiparty protocols

van den Heuvel, Bas; Pérez, Jorge A.

Published in:
Science of computer programming

DOI:
10.1016/j.scico.2022.102840

Science of Computer Programming 222 (2022) 102840

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A decentralized analysis of multiparty protocols

Bas van den Heuvel *, Jorge A. Pérez *

University of Groningen, the Netherlands

ARTICLE INFO

Article history:
Received 28 June 2021
Received in revised form 13 May 2022
Accepted 20 June 2022
Available online 30 June 2022

Keywords:
Concurrency
Process calculi
Session types
Multiparty session types
Deadlock freedom

ABSTRACT

Protocols provide the unifying glue in concurrent and distributed software today; verifying that message-passing programs conform to such governing protocols is important but difficult. Static approaches based on multiparty session types (MPST) use protocols as types to avoid protocol violations and deadlocks in programs. An elusive problem for MPST is to ensure *both* protocol conformance *and* deadlock-freedom for implementations with interleaved and delegated protocols.

We propose a decentralized analysis of multiparty protocols, specified as global types and implemented as interacting processes in an asynchronous π-calculus. Our solution rests upon two novel notions: *router processes* and *relative types*. While router processes use the global type to enable the composition of participant implementations in arbitrary process networks, relative types extract from the global type the intended interactions and dependencies between *pairs* of participants. In our analysis, processes are typed using APCP, a type system that ensures protocol conformance and deadlock-freedom with respect to *binary* protocols, developed in prior work. Our decentralized, router-based analysis enables the sound and complete transference of protocol conformance and deadlock-freedom from APCP to multiparty protocols.

1. Introduction

This paper presents a new approach to the analysis of the *protocols* that pervade concurrent and distributed software. Such protocols provide an essential unifying glue between communicating programs; ensuring that communicating programs implement protocols correctly, avoiding protocol violations and deadlocks, is an important but difficult problem. Here, we study *multiparty session types (MPST)* [38], an approach to correctness in message-passing programs that uses governing multiparty protocols as types in program verification.

As a motivating example, let us consider a *recursive authorization protocol*, adapted from an example by Scalas and Yoshida [51]. It involves three participants: a Client, a Server, and an Authorization service. Intuitively, the protocol proceeds as follows. The Server requests the Client either to *login* or to *quit* the protocol. In the case of login, the Client sends a password to the Authorization service, which then may authorize the login with the Server; subsequently, the protocol can be performed again: this is useful when, e.g., clients must renew their authorization privileges after some time. In the case of quit, the protocol ends.

MPST use *global types* to specify multiparty protocols. The authorization protocol just described can be specified by the following global type between Client (c), Server (s), and Authorization service (a):

* Corresponding authors.
E-mail addresses: b.van.den.heuvel@rug.nl (B. van den Heuvel), j.a.perez@rug.nl (J.A. Pérez).

https://doi.org/10.1016/j.scico.2022.102840

$$G_{\mathsf{auth}} = \mu X \,.\, s \twoheadrightarrow c \begin{Bmatrix} \mathsf{login} \,.\, c \twoheadrightarrow a\{\mathsf{passwd}\langle \mathsf{str}\rangle \,.\, a \twoheadrightarrow s\{\mathsf{auth}\langle \mathsf{bool}\rangle \,.\, X\}\}, \\ \mathsf{quit} \,.\, c \twoheadrightarrow a\{\mathsf{quit} \,.\, \bullet\} \end{Bmatrix} \tag{1}$$

After declaring a recursion on the variable X (μX), the global type G_{auth} stipulates that s sends to c ($s \twoheadrightarrow c$) a label login or quit. The rest of the protocol depends on this choice by s. In the login-branch, c sends to a a label passwd along with a string value ($\langle \mathsf{str}\rangle$) and a sends to s a label auth and a boolean value, after which the protocol loops to the beginning of the recursion (X). In the quit-branch, c sends to a a label quit after which the protocol ends ($\bullet$).

In MPST, participants are implemented as distributed processes that communicate asynchronously. Each process must correctly implement its corresponding portion of the protocol; these individual guarantees ensure that the interactions between processes conform to the given global type. Correctness follows from *protocol fidelity* (processes interact as stipulated by the protocol), *communication safety* (no errors or mismatches in messages), and *deadlock-freedom* (processes never get stuck waiting for each other). Ensuring that implementations satisfy these properties is a challenging problem, which is further compounded by two common and convenient features in interacting processes: *delegation* and *interleaving*. We motivate them in the context of our example:

- *Delegation,* or higher-order channel passing, can effectively express that the Client transparently reconfigures its involvement by asking another participant (say, a Password Manager) to act on its behalf;
- *Interleaving* arises when a single process implements more than one role, as in, e.g., an implementation of both the Server and the Authorization service in a sequential process.

Note that while delegation is explicitly specified in a global type, interleaving arises in its implementation as interacting processes, not in its specification.

MPST have been widely studied from foundational and applied angles [7,19,39,50,5,6,51,20,41,44]. The original theory by Honda et al. [37] defines a behavioral type system [40,3] for a π-calculus, which exploits linearity to ensure protocol fidelity and communication safety; most derived works retain this approach and target the same properties. Deadlock-freedom is hard to ensure by typing when implementations feature delegation and interleaving. In simple scenarios without interleaving and/or delegation, deadlock-freedom is easy, as it concerns a single-threaded protocol. In contrast, deadlock-freedom for processes running multiple, interleaved protocols (possibly delegated) is a much harder problem, addressed only by some advanced type systems [7,46,22].

In this paper, we tackle the problem of ensuring that networks of interacting processes correctly implement a given global type in a deadlock-free manner, while supporting delegation and interleaving. Our approach is informed by the differences between *orchestration* and *choreography*, two salient approaches to the coordination and organization of interacting processes in service-oriented paradigms [47]:

- In *orchestration*-based approaches, processes interact through a coordinator process which ensures that they all follow the protocol as intended. Quoting Van der Aalst, in an orchestration "the conductor tells everybody in the orchestra what to do and makes sure they all play in sync" [55].
- In *choreography*-based approaches, processes interact directly following the protocol without external coordination. Again quoting Van der Aalst, in a choreography "dancers dance following a global scenario without a single point of control" [55].

Specification and analysis techniques based on MPST fall under the choreography-based approach. The global type provides the protocol's specification; based on the global type, implementations for each participant interact directly with each other, without an external coordinator.

As we will see, the contrast between orchestration and choreography is relevant here because it induces a different *network topology* for interacting processes. In an orchestration, the resulting process network is *centralized*: all processes must connect to a central orchestrator process. In a choreography, the process network is *decentralized*, as processes can directly connect to each other.

Contributions We develop a new decentralized analysis of multiparty protocols.

- Here 'analysis' refers to (i) ways of specifying such protocols as interacting processes *and* (ii) techniques to verify that those processes satisfy the intended correctness properties.
- Also, aligned with the above discussion, 'decentralized' refers to the intended network topology for processes, which does not rely on an external coordinator.

Our decentralized analysis of global types enforces protocol fidelity, communication safety, and deadlock-freedom for process implementations, while uniformly supporting delegation, interleaving, and asynchronous communication.

The *key idea* of our analysis is to exploit global types to generate *router processes* (simply *routers*) that enable participant implementations to communicate directly. There is a router per participant; it is intended to serve as a "wrapper" for an individual participant's implementation. The composition of an implementation with its corresponding router is called

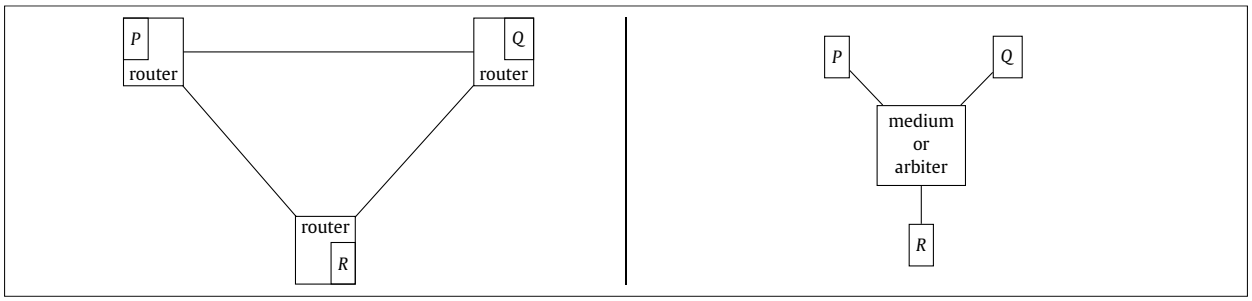

Fig. 1. Given processes P, Q, and R implementing the roles of c, s, and a, respectively, protocol G_{auth} can be realized as a choreography of routed implementations (our approach, left) and as an orchestration of implementations, with a medium or arbiter process (previous works, right).

a *routed implementation*. Collections of routed implementations can then be connected in arbitrary *process networks* that correctly realize the multiparty protocol, subsuming centralized and decentralized topologies.

Routers are *synthesized* from global types, and do not change the behavior of the participant implementations they wrap; they merely ensure that networks of routed implementations correctly behave as described by the given multiparty protocol. Returning to Van der Aalst's analogies quoted above, we may say that in our setting participant implementations are analogous to skilled but barefoot dancers, and that routers provide them with the appropriate shoes to dance without a central point of control. To make this analogy a bit more concrete, Fig. 1 (left) illustrates the decentralized process network formed by routed implementations of the participants of G_{auth}: once wrapped by an appropriate router, implementations P, Q, and R can be composed directly in a decentralized process network.

A central technical challenge in our approach is to ensure that compositions of routed implementations conform to their global type. The channels that enable the arbitrary composition of routed implementations need to be typed in accordance with the given multiparty protocol. Unfortunately, the usual notion of projection in MPST, which obtains a single participant's perspective from a global type, does not suffice: we need a local perspective that is relative to the *two participants* that the connected routed implementations represent. To this end, we introduce a new notion, *relative projection*, which isolates the exchanges of the global type that relate to pairs of participants. In the case of G_{auth}, for instance, we need three relative types, describing the protocol for a and c, for a and s, and for c and s.

A derived challenge is that when projecting a global type onto a pair of participants, it is possible to encounter *non-local choices*: choices by other participants that affect the protocol between the two participants involved in the projection. To handle this, relative projection explicitly records non-local choices in the form of *dependencies*, which inform the projection's participants that they need to coordinate on the results of the non-local choices.

To summarize, our decentralized analysis of global types relies on three intertwined novel notions:

- **Routers** that wrap participant implementations in order to enable their composition in arbitrary network topologies, whilst guaranteeing that the resulting process networks correctly follow the given global type in a deadlock-free manner.
- **Relative Types** that type the channels between routed implementations, obtained by means of a new notion of projection of global types onto pairs of participants.
- **Relative Projection** and **Dependencies** that make it explicit in relative types that participants need to coordinate on non-local choices.

The key ingredients of our decentralized analysis for G_{auth} are jointly depicted in Fig. 2.

With respect to prior analyses of multiparty protocols, a distinguishing feature of our work is its natural support of decentralized process networks, as expected in a choreography-based approach. Caires and Pérez [12] connect participant implementations through a central coordinator, called *medium process*. This medium process is generated from a global type, and intervenes in all exchanges to ensure that the participant implementations follow the multiparty protocol. The composition of the medium with the participant implementations can then be analyzed using a type system for binary sessions. In a similar vein, Carbone et al. [17] define a type system in which they use global types to validate choreographies of participant implementations. Their analysis of protocol implementations—in particular, deadlock-freedom—relies on encodings into another type system where participant implementations connect to a central coordinator, called the *arbiter process*. Similar to mediums, arbiters are generated from the global type to ensure that participant implementations follow the protocol as intended. Both these approaches are clear examples of orchestration, and thus do not support decentralized network topologies.

To highlight the differences between our decentralized analysis and prior approaches, compare the choreography of routed implementations in Fig. 1 (left) with an implementation of G_{auth} in the style of Caires and Pérez and of Carbone et al., given in Fig. 1 (right). These prior works rely on orchestration because the type systems they use for verifying process implementations restrict connections between processes: they only admit a form of process composition that makes it

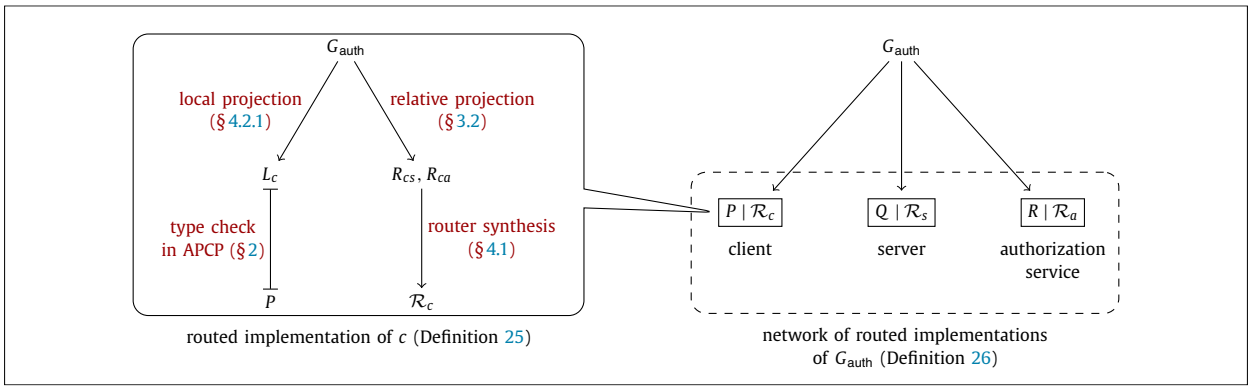

Fig. 2. Decentralized analysis of G_{auth} into a network of routed implementations. The definition of G_{auth} contains message types. Focusing on the client c (on the left), L_c denotes a session type, whereas R_{cs} and R_{ca} are relative types with respect to the server and the authorization service, respectively.

impossible to simultaneously connect three or more participant implementations [26,25]. In this paper, we overcome this obstacle by relying on APCP (Asynchronous Priority-based Classical Processes) [54], a type system that allows for more general forms of process composition. By using annotations on types, APCP prevents *circular dependencies*, i.e., cyclically connected processes that are stuck waiting for each other. This is how our approach supports networks of routed participants in both centralized and decentralized topologies, thus subsuming choreography and orchestration approaches.

Outline This paper is structured as follows. Next, Section 2 recalls APCP as introduced by Van den Heuvel and Pérez [54] and summarizes the correctness properties for asynchronous processes derived from typing. The following three sections develop and illustrate our contributions:

- Section 3 introduces relative types and relative projection, and defines *well-formed* global types, a class of global types that includes protocols with non-local choices.
- Section 4 introduces the synthesis of routers. A main result is their typability in APCP (Theorem 11). We establish deadlock-freedom for networks of routed implementations (Theorem 18), which we transfer to multiparty protocols via an operational correspondence result (Theorems 19 and 23). Moreover, we show that our approach strictly generalizes prior analyses based on centralized topologies (Theorem 27).
- Section 5 demonstrates our contributions with a full development of the routed implementations for G_{auth}, and an example of the flexible support for *delegation* and *interleaving* enabled by our router-based approach and APCP.

We discuss further related works in §6 and conclude the paper in §7. We use colors to improve readability—for interpretation of the colors, the reader is referred to the web version of this article.

2. APCP: asynchronous processes, deadlock-free by typing

We recall APCP as defined by Van den Heuvel and Pérez [54]. APCP is a type system for asynchronous π-calculus processes (with non-blocking outputs) [36,9], with support for recursion and cyclic connections. In this type system, channel endpoints are assigned linear types that represent two-party (binary) *session types* [35]. Well-typed APCP processes preserve typing (Theorem 2) and are deadlock-free (Theorem 5).

At its basis, APCP combines Dardha and Gay's Priority-based Classical Processes (PCP) [23] with DeYoung et al.'s continuation-passing semantics for asynchrony [29], and adds recursion, inspired by the work of Toninho et al. [52]. We refer the interested reader to the work by Van den Heuvel and Pérez [54] for a motivation of design choices and proofs of results.

Process syntax We write $x, y, z, \ldots$ to denote (channel) *endpoints* (also known as *names*), and write $\tilde{x}, \tilde{y}, \tilde{z}, \ldots$ to denote sequences of endpoints. With a slight abuse of notation, we sometimes write $x_i \in \tilde{x}$ to refer to a specific element in the sequence $\tilde{x}$. Also, we write $i, j, k, \ldots$ to denote *labels* for choices and $I, J, K, \ldots$ to denote sets of labels. We write $X, Y, \ldots$ to denote *recursion variables*, and $P, Q, \ldots$ to denote processes.

Fig. 3 (top) gives the syntax of processes. The output action $x[y, z]$ sends a message y (an endpoint) and a continuation endpoint z along x. The input prefix $x(y, z) \,.\, P$ blocks until a message and a continuation endpoint are received on x (referred to in P as the placeholders y and z, respectively), binding y and z in P. The selection action $x[z] \triangleleft i$ sends a label i and a continuation endpoint z along x. The branching prefix $x(z) \triangleright \{i : P_i\}_{i \in I}$ blocks until it receives a label $i \in I$ and a continuation endpoint (referred to in P_i as the placeholder z) on x, binding z in each P_i. Restriction $(\boldsymbol{\nu} xy)P$ binds x and y in P, thus declaring them as the two endpoints of the same channel and enabling communication, as in Vasconcelos [56].

Process syntax:

$$
\begin{array}{llllll}
P, Q ::= x[y,z] & \text{output} & \mid\ x(y,z)\,.\,P & \text{input} & & \\
\mid\ x[z] \triangleleft i & \text{selection} & \mid\ x(z) \triangleright \{i : P_i\}_{i\in I} & \text{branching} & \mid\ (\boldsymbol{\nu} xy)P & \text{restriction} \\
\mid\ (P \mid Q) & \text{parallel} & \mid\ \mathbf{0} & \text{inaction} & \mid\ x \leftrightarrow y & \text{forwarder} \\
\mid\ \mu X(\tilde{z})\,.\,P & \text{recursive loop} & \mid\ X\langle \tilde{z}\rangle & \text{recursive call} & &
\end{array}
$$

Structural congruence:

$$
\begin{array}{lll}
P \equiv_\alpha P' \implies & P \equiv P' & x \leftrightarrow y \equiv y \leftrightarrow x \\
& P \mid Q \equiv Q \mid P & (\boldsymbol{\nu} xy)x \leftrightarrow y \equiv \mathbf{0} \\
& P \mid \mathbf{0} \equiv P & P \mid (Q \mid R) \equiv (P \mid Q) \mid R \\
x, y \notin \mathrm{fn}(P) \implies & P \mid (\boldsymbol{\nu} xy)Q \equiv (\boldsymbol{\nu} xy)(P \mid Q) & (\boldsymbol{\nu} xy)\mathbf{0} \equiv \mathbf{0} \\
|\tilde{z}| = |\tilde{y}| \implies & \mu X(\tilde{z})\,.\,P \equiv P\big\{(\mu X(\tilde{y})\,.\,P\{\tilde{y}/\tilde{z}\})/X\langle\tilde{y}\rangle\big\} & (\boldsymbol{\nu} xy)P \equiv (\boldsymbol{\nu} yx)P \\
& & (\boldsymbol{\nu} xy)(\boldsymbol{\nu} zw)P \equiv (\boldsymbol{\nu} zw)(\boldsymbol{\nu} xy)P
\end{array}
$$

Reduction:

$$
\begin{array}{lll}
\rightarrow_{\mathrm{ID}} & z, y \neq x \implies & (\boldsymbol{\nu} yz)(x \leftrightarrow y \mid P) \longrightarrow P\{x/z\} \\
\rightarrow_{\otimes\parr} & & (\boldsymbol{\nu} xy)(x[a,b] \mid y(v,z)\,.\,P) \longrightarrow P\{a/v, b/z\} \\
\rightarrow_{\oplus\&} & j \in I \implies & (\boldsymbol{\nu} xy)(x[b] \triangleleft j \mid y(z) \triangleright \{i : P_i\}_{i\in I}) \longrightarrow P_j\{b/z\}
\end{array}
$$

$$
\frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q} \rightarrow_{\equiv} \qquad \frac{P \longrightarrow Q}{(\boldsymbol{\nu} xy)P \longrightarrow (\boldsymbol{\nu} xy)Q} \rightarrow_{\nu} \qquad \frac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R} \rightarrow_{\mid}
$$

Fig. 3. Definition of APCP's process language.

The process $P \mid Q$ denotes the parallel composition of P and Q. The process $\mathbf{0}$ denotes inaction. The forwarder process $x \leftrightarrow y$ is a primitive copycat process that links together x and y. The prefix $\mu X(\tilde{z})\,.\,P$ defines a recursive loop, binding occurrences of X in P; the endpoints $\tilde{z}$ form a context for P. The recursive call $X\langle\tilde{z}\rangle$ loops to its corresponding μX, providing the endpoints $\tilde{z}$ as context. We only consider contractive recursion, disallowing processes with subexpressions of the form $\mu X_1(\tilde{z}) \ldots \mu X_n(\tilde{z})\,.\,X_1\langle\tilde{z}\rangle$.

Endpoints and recursion variables are free unless otherwise stated (i.e., unless they are bound somehow). We write $\mathrm{fn}(P)$ and $\mathrm{frv}(P)$ for the sets of free names and free recursion variables of P, respectively. Also, we write $P\{x/y\}$ to denote the capture-avoiding substitution of the free occurrences of y in P for x. The notation $P\{\mu X(\tilde{y})\,.\,P'/X\langle\tilde{y}\rangle\}$ denotes the substitution of occurrences of recursive calls $X\langle\tilde{y}\rangle$ in P with the recursive loop $\mu X(\tilde{y})\,.\,P'$, which we call *unfolding* recursion. We write sequences of substitutions $P\{x_1/y_1\}\ldots\{x_n/y_n\}$ as $P\{x_1/y_1, \ldots, x_n/y_n\}$.

In an output $x[y,z]$, both y and z are free; they can be bound to a continuation process using parallel composition and restriction, as in $(\boldsymbol{\nu} ya)(\boldsymbol{\nu} zb)(x[y,z] \mid P_{a,b})$. The same applies to selection $x[z] \triangleleft i$. We introduce useful notations that elide the restrictions and continuation endpoints:

Notation 1 *(Derivable actions and prefixes).* We use the following syntactic sugar:

$$
\begin{array}{ll}
\overline{x}[y] \cdot P := (\boldsymbol{\nu} ya)(\boldsymbol{\nu} zb)(x[a,b] \mid P\{z/x\}) & \overline{x} \triangleleft \ell \cdot P := (\boldsymbol{\nu} zb)(x[b] \triangleleft \ell \mid P\{z/x\}) \\
x(y)\,.\,P := x(y,z)\,.\,P\{z/x\} & x \triangleright \{i : P_i\}_{i\in I} := x(z) \triangleright \{i : P_i\{z/x\}\}_{i\in I}
\end{array}
$$

Note the use of '$\cdot$' instead of '$.$' in output and selection to stress that they are non-blocking.

Operational semantics We define a reduction relation for processes ($P \longrightarrow Q$) that formalizes how complementary actions on connected endpoints may synchronize. As usual for π-calculi, reduction relies on *structural congruence* ($P \equiv Q$), which relates processes with minor syntactic differences; it is the smallest congruence on the syntax of processes (Fig. 3 (top)) satisfying the axioms in Fig. 3 (middle).

Structural congruence defines the following properties of our process language. Processes are equivalent up to α-equivalence. Parallel composition is associative and commutative, with unit $\mathbf{0}$. The forwarder process is symmetric, and equivalent to inaction if both endpoints are bound together through restriction. A parallel process may be moved into or

Table 1
Session types and their associated endpoint behaviors (cf. Definition 1).

Session Type	Endpoint Behavior
$A \otimes^{\circ} B$	output an endpoint of type A, then behave as B
$A \parr^{\circ} B$	input an endpoint of type A, then behave as B
$\oplus^{\circ}\{i : A_i\}_{i \in I}$	select a label $i \in I$, then behave as A_i
$\&^{\circ}\{i : A_i\}_{i \in I}$	receive a choice for a label $i \in I$, then behave as A_i
$\bullet$	closed session; no behavior

out of a restriction as long as the bound channels do not occur free in the moved process: this is *scope inclusion* and *scope extrusion*, respectively. Restrictions on inactive processes may be dropped, and the order of endpoints in restrictions and of consecutive restrictions does not matter. Finally, a recursive loop is equivalent to its unfolding, replacing any recursive calls with copies of the recursive loop, where the call's endpoints are pairwise substituted for the contextual endpoints of the loop.

As we will see next, the semantics of APCP is closed under structural congruence. This means that processes are *equi-recursive*; however, APCP's typing discipline (described next) treats recursive types as *iso-recursive* (see, e.g., Pierce [48]).

We can now define our reduction relation. We define the reduction relation $P \longrightarrow Q$ by the axioms and closure rules in Fig. 3 (bottom). Presentations of Curry-Howard interpretations of linear logic often include *commuting conversions* (such as [14,57,23,29]), which allow pulling prefixes on free channels out of restrictions; they are not necessary for deadlock-freedom in APCP, so we do not include them.

Rule $\rightarrow_{\mathrm{Id}}$ implements the forwarder as a substitution. Rule $\rightarrow_{\otimes\parr}$ synchronizes an output and an input on connected endpoints and substitutes the message and continuation endpoint. Rule $\rightarrow_{\oplus\&}$ synchronizes a selection and a branch: the received label determines the continuation process, substituting the continuation endpoint appropriately. Rules $\rightarrow_{\equiv}$, $\rightarrow_{\nu}$, and $\rightarrow_{|}$ close reduction under structural congruence, restriction, and parallel composition, respectively.

Notation 2 *(Reductions).* We write $\longrightarrow^*$ for the reflexive, transitive closure of $\longrightarrow$. Also, we write $P \longrightarrow^{\star} Q$ if $P \longrightarrow^* Q$ in a finite number of steps, and $P \not\longrightarrow^* Q$ for the non-existence of a series of reductions from P to Q.

Session types APCP types processes by assigning binary session types to channel endpoints. Following Curry-Howard interpretations, we present session types as linear logic propositions (cf., e.g., Caires et al. [15], Wadler [57], Caires and Pérez [13], and Dardha and Gay [23]). We extend these propositions with recursion and *priority* annotations on connectives. Intuitively, actions typed with lower priority should be performed before those with higher priority.

We write $\mathrm{o}, \kappa, \pi, \rho, \ldots$ to denote priorities, and ω to denote the ultimate priority that is greater than all other priorities and cannot be increased further. That is, $\forall t \in \mathbb{N}.\ \omega > t$ and $\forall t \in \mathbb{N}.\ \omega + t = \omega$.

Definition 1 *(Session types).* The following grammar defines the syntax of *session types* A, B. Let $\mathrm{o} \in \mathbb{N} \cup \{\omega\}$.

$$A, B ::= A \otimes^{\circ} B \mid A \parr^{\circ} B \mid \oplus^{\circ}\{i : A\}_{i \in I} \mid \&^{\circ}\{i : A\}_{i \in I} \mid \bullet \mid \mu X\,.\,A \mid X$$

Table 1 gives session types and the behavior that is expected of an endpoint with each type (recursive types entail a communication behavior only after unfolding). Note that $\bullet$ does not require a priority, as closed endpoints do not exhibit behavior and thus are non-blocking. We define $\bullet$ as a single, self-dual type for closed endpoints (cf. Caires [11] and Atkey et al. [4]).

Type $\mu X\,.\,A$ denotes a recursive type, in which A may contain occurrences of the recursion variable X. As customary, μ is a binder: it induces the standard notions of α-equivalence, substitution (denoted $A\{B/X\}$), and free recursion variables (denoted $\mathrm{frv}(A)$). We work with tail-recursive, contractive types, disallowing types of the form $\mu X_1 \ldots \mu X_n\,.\,X_1$ and $\mu X\,.\,X \otimes^{\circ} A$. Recursive types are treated iso-recursively: there will be an explicit typing rule that unfold recursive types, and recursive types are not equal to their unfolding. We postpone formalizing the unfolding of recursive types, as it requires additional definitions to ensure consistency of priorities upon unfolding.

Duality, the cornerstone of session types and linear logic, ensures that the two endpoints of a channel have matching actions. Furthermore, dual types must have matching priority annotations. The following inductive definition of duality suffices for our tail-recursive types (cf. Gay et al. [33]).

Definition 2 *(Duality).* The *dual* of session type A, denoted $\overline{A}$, is defined inductively as follows:

$$\overline{A \otimes^{\circ} B} := \overline{A} \parr^{\circ} \overline{B} \qquad \overline{\oplus^{\circ}\{i : A_i\}_{i \in I}} := \&^{\circ}\{i : \overline{A_i}\}_{i \in I} \qquad \overline{\bullet} := \bullet \qquad \overline{\mu X\,.\,A} := \mu X\,.\,\overline{A}$$
$$\overline{A \parr^{\circ} B} := \overline{A} \otimes^{\circ} \overline{B} \qquad \overline{\&^{\circ}\{i : A_i\}_{i \in I}} := \oplus^{\circ}\{i : \overline{A_i}\}_{i \in I} \qquad \overline{X} := X$$

The priority of a type is determined by the priority of the type's outermost connective:

Definition 3 *(Priorities).* For session type A, $\mathsf{pr}(A)$ denotes its *priority*:

$$\mathsf{pr}(A \otimes^{\mathsf{o}} B) := \mathsf{pr}(A \parr^{\mathsf{o}} B) := \mathsf{o} \qquad \mathsf{pr}(\mu X\,.\,A) := \mathsf{pr}(A)$$
$$\mathsf{pr}(\oplus^{\mathsf{o}}\{i : A_i\}_{i\in I}) := \mathsf{pr}(\&^{\mathsf{o}}\{i : A_i\}_{i\in I}) := \mathsf{o} \qquad \mathsf{pr}(\bullet) := \mathsf{pr}(X) := \omega$$

The priority of $\bullet$ and X is ω: they denote "final", non-blocking actions of protocols. Although $\otimes$ and $\oplus$ also denote non-blocking actions, their priority is not constant: duality ensures that the priority for $\otimes$ (resp. $\oplus$) matches the priority of a corresponding $\parr$ (resp. &), which denotes a blocking action.

We now turn to formalizing the unfolding of recursive types. Recall the intuition that actions typed with lower priority should be performed before those with higher priority. Based on this rationale, we observe that the unfolding of the recursive type $\mu X\,.\,A$ should not result in $A\{\mu X\,.\,A/X\}$, as usual, but that the priorities of the unfolded type should be *increased*. This is because the actions related to the unfolded recursion should be performed *after* the prefix. For example, consider the recursive type $\mu X\,.\,A \parr^0 X$. If we unfold this type without increasing the priority, we would obtain $A \parr^0 (\mu X\,.\,A \parr^0 X)$, a type in which the priorities no longer determine a global ordering between the two inputs. By increasing the priority in the unfolded type as in, e.g., $A \parr^0 (\mu X\,.\,A \parr^1 X)$, a global ordering is preserved.

We make this intuition precise by defining the *lift* of priorities in types:

Definition 4 *(Lift).* For proposition A and $t \in \mathbb{N}$, we define $\uparrow^t A$ as the *lift* operation:

$$\uparrow^t(A \otimes^{\mathsf{o}} B) := (\uparrow^t A) \otimes^{\mathsf{o}+t} (\uparrow^t B) \qquad \uparrow^t(\oplus^{\mathsf{o}}\{i : A_i\}_{i\in I}) := \oplus^{\mathsf{o}+t}\{i : \uparrow^t A_i\}_{i\in I} \qquad \uparrow^t \bullet := \bullet$$
$$\uparrow^t(A \parr^{\mathsf{o}} B) := (\uparrow^t A) \parr^{\mathsf{o}+t} (\uparrow^t B) \qquad \uparrow^t(\&^{\mathsf{o}}\{i : A_i\}_{i\in I}) := \&^{\mathsf{o}+t}\{i : \uparrow^t A_i\}_{i\in I}$$
$$\uparrow^t(\mu X\,.\,A) := \mu X\,.\,(\uparrow^t A) \qquad \uparrow^t X := X$$

Henceforth, the unfolding of $\mu X\,.\,A$ is $A\{\mu X\,.\,(\uparrow^t A)/X\}$, denoted $\mathsf{unfold}^t(\mu X\,.\,A)$, where $t \in \mathbb{N}$ depends on the highest priority of the types occurring in a typing context. We recall that we do not consider types to be equi-recursive: recursive types are not equal to their unfolding. Recursive types can only be unfolded by typing rules, discussed next.

We now define the highest priority of a type:

Definition 5 *(Highest priority).* For session type A, $\max_{\mathsf{pr}}(A)$ denotes its *highest priority*:

$$\max_{\mathsf{pr}}(A \otimes^{\mathsf{o}} B) := \max_{\mathsf{pr}}(A \parr^{\mathsf{o}} B) := \max(\max_{\mathsf{pr}}(A), \max_{\mathsf{pr}}(B), \mathsf{o})$$
$$\max_{\mathsf{pr}}(\oplus^{\mathsf{o}}\{i : A_i\}_{i\in I}) := \max_{\mathsf{pr}}(\&^{\mathsf{o}}\{i : A_i\}_{i\in I}) := \max(\max_{i\in I}(\max_{\mathsf{pr}}(A_i)), \mathsf{o})$$
$$\max_{\mathsf{pr}}(\mu X\,.\,A) := \max_{\mathsf{pr}}(A)$$
$$\max_{\mathsf{pr}}(\bullet) := \max_{\mathsf{pr}}(X) := 0$$

Notice how the highest priority of $\bullet$ and X is 0, in contrast to their priority (as given by Definition 3): they do not contribute to the increase in priority needed for unfolding recursive types.

Typing rules The typing rules of APCP ensure that actions with lower priority are performed before those with higher priority (cf. Dardha and Gay [23]). To this end, they enforce the following laws:

1. an action with priority o must be prefixed only by inputs and branches with priority strictly smaller than o—this law does not hold for output and selection, as they are not prefixes;
2. dual actions leading to a synchronization must have equal priorities (cf. Def. 1).

Judgments are of the form $P \vdash \Omega; \Gamma$, where:

- P is a process;
- Γ is a context that assigns types to channels $(x : A)$;
- Ω is a context that assigns tuples of types to recursion variables $(X : (A, B, \ldots))$.

A judgment $P \vdash \Omega; \Gamma$ then means that P can be typed in accordance with the type assignments for names recorded in Γ and the recursion variables in Ω. Intuitively, the latter context ensures that the context endpoints between recursive definitions and calls concur. Both contexts Γ and Ω obey *exchange*: assignments may be silently reordered. Γ is *linear*, disallowing *weakening* (i.e., all assignments must be used) and *contraction* (i.e., assignments may not be duplicated). Ω allows weakening and contraction, because a recursive definition may be called *zero or more* times.

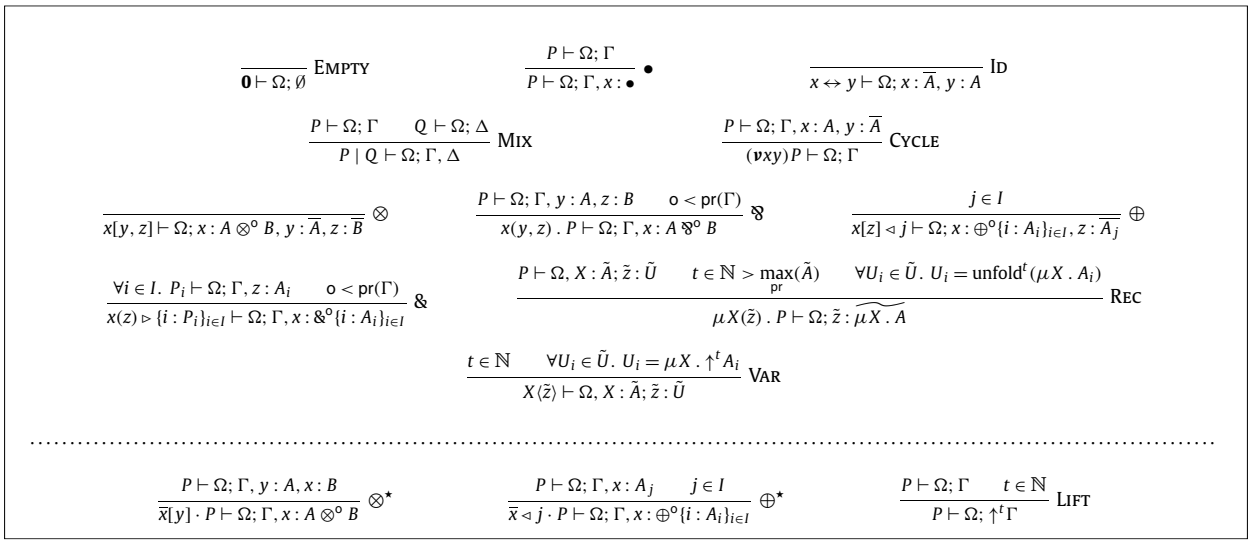

Fig. 4. The typing rules of APCP (top) and admissible rules (bottom).

The empty context is written $\emptyset$. In writing $\Gamma, x : A$ (or similarly for Ω) we assume that $x \notin \mathsf{dom}(\Gamma)$. We write $\uparrow^{t}\Gamma$ to denote the component-wise extension of lift (Definition 4) to typing contexts. Also, we write $\mathsf{pr}(\Gamma)$ to denote the least priority of all types in Γ (Definition 3). An assignment $\tilde{z} : \tilde{A}$ means $z_1 : A_1, \ldots, z_k : A_k$.

Fig. 4 (top) gives the typing rules. We describe the typing rules from a *bottom-up* perspective. Rule Empty types an inactive process with no endpoints. Rule • silently removes a closed endpoint to the typing context. Rule Id types forwarding between endpoints of dual type. Rule Mix types the parallel composition of two processes that do not share assignments on the same endpoints. Rule Cycle types a restriction, where the two restricted endpoints must be of dual type. Note that a single application of Mix followed by Cycle coincides with the usual Rule Cut in type systems based on linear logic [14,57]. Rule ⊗ types an output action; this rule does not have premises to provide a continuation process, leaving the free endpoints to be bound to a continuation process using Mix and Cycle. Similarly, Rule ⊕ types an unbounded selection action. Priority checks are confined to Rules ⅋ and &, which type an input and a branching prefix, respectively. In both cases, the used endpoint's priority must be lower than the priorities of the other types in the continuation's typing context.

Rule Rec types a recursive definition by introducing a recursion variable to the recursion context whose tuple of types concurs with the contents of the recursive types in the typing context, where contractivity is guaranteed by requiring that the eliminated recursion variable may not occur unguarded in each of the context's types. At the same time, the recursive types in the context are unfolded, and their priorities are lifted by a common value, denoted t in the rule, that must be greater than the highest priority occurring in the original types (cf. Definition 5). Using a "common lifter", i.e., lifting the priorities of all types by the same amount is crucial: it maintains the relation between the priorities of the types in the context.

Rule Var types a recursive call on a variable in the recursive context. The rule requires that all the types in the context are recursive on the recursion variable called, and that the types inside the recursive definitions concur with the respective types assigned to the recursion variable in the recursive context. As mentioned before, the types associated to the introduced (and consequently eliminated) recursion variable are crucial in ensuring that a recursion is called with endpoints of the same type as required by its definition.

The binding of output and selection actions to continuation processes (Notation 1) is derivable in APCP. The corresponding typing rules in Fig. 4 (bottom) are admissible using Mix and Cycle. Note that it is not necessary to include rules for the sugared input and branching in Notation 1, because they rely on α-renaming only. Fig. 4 (bottom) also includes an admissible Rule Lift that lifts a process' priorities.

The following result assures that, given a type, we can construct a process with an endpoint typable with the given type:

Proposition 1. *Given a type A, there exists a P such that $P \vdash \Omega; x : A$.*

Proof. We inductively define a function $\mathrm{char}^{x}(A)$ that, given a type A and an endpoint x, constructs a process that performs the behavior described by A:

$$\mathrm{char}^{x}(A \otimes^{o} B) := \overline{x}[y] \cdot (\mathrm{char}^{y}(A) \mid \mathrm{char}^{x}(B)) \qquad \mathrm{char}^{x}(\oplus^{o}\{i : A_i\}_{i \in I}) := \overline{x} \triangleleft j \cdot \mathrm{char}^{x}(A_j) \quad [\text{any } j \in I]$$

$$\mathrm{char}^{x}(A \parr^{o} B) := x(y) . (\mathrm{char}^{y}(A) \mid \mathrm{char}^{x}(B)) \qquad \mathrm{char}^{x}(\&^{o}\{i : A_i\}_{i \in I}) := x \triangleright \{i : \mathrm{char}^{x}(A_i)\}_{i \in I}$$

$$\mathrm{char}^x(\bullet) := \mathbf{0} \qquad \mathrm{char}^x(\mu X\,.\,A) := \mu X(x)\,.\,\mathrm{char}^x(A) \qquad \mathrm{char}^x(X) := X\langle x\rangle$$

For finite types, we have: $\mathrm{char}^x(A) \vdash \emptyset; x : A$. For simplicity, we omit details about recursive types, which require unfolding. For closed, recursive types, we have: $\mathrm{char}^x(\mu X\,.\,A) \vdash \emptyset; x : \mu X\,.\,A$. □

Type preservation Well-typed processes satisfy protocol fidelity, communication safety, and deadlock-freedom. The first two properties follow directly from *type preservation* (also known as *subject reduction*), which ensures that reduction preserves typing. In contrast to Caires and Pfenning [14] and Wadler [57], where type preservation corresponds to the elimination of (top-level) applications of rule Cut, in APCP it corresponds to the more general elimination of (top-level) applications of rule Cycle.

Because reduction is closed under structural congruence, type preservation relies on *subject congruence*: structural congruence preserves typing. The structural congruence rule that unfolds recursive definitions requires care, because the types of the unfolded process are also unfolded. Hence, type preservation holds *up to unfolding*. We formalize this with the relation $(P;\Gamma) \precsim \Gamma'$, which denotes that Γ and Γ' are equal up to (un)folding of recursive types consistent with the typing of P under Γ:

Definition 6. We define an asymmetrical relation between a process / typing pair $(P;\Gamma)$ and a typing context Γ', denoted $(P;\Gamma) \precsim \Gamma'$. The relation is defined inductively, with the following being the most important rules relating unfolded and non-unfolded types:

$$\frac{(P\{(\mu X(\tilde{y})\,.\,P\{\tilde{y}/\tilde{z}\})/X\langle\tilde{y}\rangle\};\tilde{z} : \tilde{U}) \precsim \tilde{z} : \tilde{U}' \qquad \forall U_i' \in \tilde{U}'.\ U_i' = \mathrm{unfold}^t(\mu X\,.\,A_i')}{(P\{(\mu X(\tilde{y})\,.\,P\{\tilde{y}/\tilde{z}\})/X\langle\tilde{y}\rangle\};\tilde{z} : \tilde{U}) \precsim \tilde{z} : \widetilde{\mu X\,.\,A'}}\ \textsc{Fold}$$

$$\frac{(\mu X(\tilde{z})\,.\,P;\tilde{z} : \widetilde{\mu X\,.\,A}) \precsim \tilde{z} : \widetilde{\mu X\,.\,A'} \qquad \forall U_i' \in \tilde{U}'.\ U_i' = \mathrm{unfold}^t(\mu X\,.\,A_i')}{(\mu X(\tilde{z})\,.\,P;\tilde{z} : \widetilde{\mu X\,.\,A}) \precsim \tilde{z} : \tilde{U}'}\ \textsc{Unfold}$$

The other rules follow the typing rules in Fig. 4; the following is a selection of rules:

$$\frac{}{(\mathbf{0};\emptyset) \precsim \emptyset}\ \textsc{Empty} \qquad \frac{(P;\Gamma) \precsim \Gamma' \qquad (Q\,;\Delta) \precsim \Delta'}{(P \mid Q\,;\Gamma,\Delta) \precsim \Gamma',\Delta'}\ \textsc{Mix} \qquad \frac{(P;\Gamma, y : A, z : B) \precsim \Gamma', y : A', z : B'}{(x(y,z)\,.\,P;\Gamma, x : A \parr^{\circ} B) \precsim \Gamma', x : A' \parr^{\circ} B'}\ \parr$$

We write $(P;\Gamma) \cong (Q\,;\Gamma')$ if $(P;\Gamma) \precsim \Gamma'$ and $(Q\,;\Gamma') \precsim \Gamma$.

Theorem 2 *(Type preservation). If $P \vdash \Omega;\Gamma$ and $P \longrightarrow Q$, then there exists Γ' such that $Q \vdash \Omega;\Gamma'$ and $(P;\Gamma) \cong (Q\,;\Gamma')$.*

Deadlock-freedom The deadlock-freedom result for APCP adapts that for PCP [23]. As mentioned before, binding asynchronous outputs and selections to continuations involves additional, low-level uses of Cycle, which we cannot eliminate through process reduction. Therefore, top-level deadlock-freedom (referred to as *progress*) holds for *live processes* (Theorem 4). A process is live if it is equivalent to a restriction on *active names* that perform unguarded actions. This way, e.g., in $x[y,z]$ the name x is active, but y and z are not. We additionally need a notion of *evaluation context*, under which reducible forwarders may occur.

Definition 7 *(Active names).* The *set of active names* of P, denoted $\mathrm{an}(P)$, contains the (free) names that are used for unguarded actions (output, input, selection, branching):

$$\begin{array}{lll}
\mathrm{an}(x[y,z]) := \{x\} & \mathrm{an}(x(y,z)\,.\,P) := \{x\} & \mathrm{an}(\mathbf{0}) := \emptyset \\
\mathrm{an}(x[z] \lhd j) := \{x\} & \mathrm{an}(x(z) \rhd \{i : P_i\}_{i\in I}) := \{x\} & \mathrm{an}(x \leftrightarrow y) := \emptyset \\
\mathrm{an}(P \mid Q) := \mathrm{an}(P) \cup \mathrm{an}(Q) & \mathrm{an}(\mu X(\tilde{x})\,.\,P) := \mathrm{an}(P) & \\
\mathrm{an}((\boldsymbol{\nu} xy)P) := \mathrm{an}(P) \setminus \{x,y\} & \mathrm{an}(X\langle\tilde{x}\rangle) := \emptyset &
\end{array}$$

Definition 8 *(Evaluation context).* Evaluation contexts $\mathcal{E}$ are defined by the following grammar:

$$\mathcal{E} ::= [\,] \mid \mathcal{E} \mid P \mid (\boldsymbol{\nu} xy)\mathcal{E} \mid \mu X(\tilde{x})\,.\,\mathcal{E}$$

We write $\mathcal{E}[P]$ to denote the process obtained by replacing the hole $[\,]$ in $\mathcal{E}$ by P.

Definition 9 *(Live process).* A process P is *live*, denoted $\mathrm{live}(P)$, if

1. there are names x, y and process P' such that $P \equiv (\boldsymbol{\nu} xy)P'$ with $x, y \in \mathrm{an}(P')$, or
2. there are names x, y, z and process P' such that $P \equiv \mathcal{E}[(\boldsymbol{\nu} yz)(x \leftrightarrow y \mid P')]$.

We additionally need to account for recursion: as recursive definitions do not entail reductions, we must fully unfold them before eliminating Cycles:

Lemma 3 *(Unfolding). If $P \vdash \Omega; \Gamma$, then there is a process $P^\star$ such that $P^\star \equiv P$ and $P^\star$ is not of the form $\mu X(\tilde{z})\,.\,Q$.*

Deadlock-freedom, given next, states that typable processes that are live can reduce. It follows from an analysis of the priorities in the typing of the process, which makes it possible to find a pair of non-blocked, parallel, dual actions on connected endpoints, such that a communication can occur. The analysis also considers the possibility that a blocking action is on an endpoint which is not connected (i.e., the endpoint is free), in which case a commuting conversion can be performed. Confer the full proof by Van den Heuvel and Pérez [54, Theorem 7] for more details.

Theorem 4 *(Progress). If $P \vdash \emptyset; \Gamma$ and $\mathrm{live}(P)$, then there is a process Q such that $P \longrightarrow Q$.*

We now state the deadlock-freedom result formalized by Van den Heuvel and Pérez [54]. Following, e.g., Caires and Pfenning [14] and Dardha and Gay [23], it concerns processes typable under empty contexts.

Theorem 5 *(Deadlock-freedom [54]). If $P \vdash \emptyset; \emptyset$, then either $P \equiv \mathbf{0}$ or $P \longrightarrow Q$ for some Q.*

Fairness Processes typable under empty contexts are not only deadlock-free, they are *fair*: for each endpoint in the process, we can eventually observe a reduction involving that endpoint. To formalize this property, we define *labeled reductions*, which expose details about a communication:

Definition 10 *(Labeled reductions).* Consider the labels

$$\alpha ::= x \leftrightarrow y \;\mid\; x\rangle y : a \;\mid\; x\rangle y : \ell \qquad \text{(forwarding, output/input, selection/branching)}$$

where each label has subjects x and y. The *labeled reduction* $P \xrightarrow{\alpha} Q$ is defined by the following rules:

$$(\boldsymbol{\nu} yz)(x \leftrightarrow y \mid P) \xrightarrow{x \leftrightarrow y} P\{x/z\} \qquad (\boldsymbol{\nu} xy)(x[a,b] \mid y(v,z)\,.\,P) \xrightarrow{x\rangle y:a} P\{a/v, b/z\}$$

$$(\boldsymbol{\nu} xy)(x[b] \triangleleft j \mid y(z) \triangleright \{i : P_i\}_{i\in I}) \xrightarrow{x\rangle y:j} P_j\{b/z\} \quad (\text{if } j \in I)$$

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

Proposition 6. *For any P and P', $P \longrightarrow P'$ if and only if there exists a label α such that $P \xrightarrow{\alpha} P'$.*

Proof. Immediate by definition, for each reduction in Fig. 3 (bottom) corresponds to a labeled reduction, and vice versa. □

Fairness states that processes typable under empty contexts have at least one finite reduction sequence ($\longrightarrow^\star$) that enables a labeled reduction involving a *pending* endpoint—an endpoint that occurs as the subject of an action, and is not bound by input or branching (see below). Clearly, the typed process may have other reduction sequences, not necessarily finite. Confer the full proof by Van den Heuvel and Pérez [54, Theorem 11] for more details.

Definition 11 *(Pending names).* Given a process P, we define the set of *pending names* of P, denoted $\mathrm{pn}(P)$, as follows:

$$\mathrm{pn}(x[y,z]) := \{x\} \qquad \mathrm{pn}(x(y,z).P) := \{x\} \cup (\mathrm{pn}(P) \setminus \{y,z\}) \qquad \mathrm{pn}(\mathbf{0}) := \emptyset$$

$$\mathrm{pn}(x[z] \triangleleft j) := \{x\} \qquad \mathrm{pn}(x(z) \triangleright \{i : P_i\}_{i\in I}) := \{x\} \cup (\textstyle\bigcup_{i\in I} \mathrm{pn}(P_i) \setminus \{z\}) \qquad \mathrm{pn}(x \leftrightarrow y) := \{x, y\}$$

$$\mathrm{pn}(P \mid Q) := \mathrm{pn}(P) \cup \mathrm{pn}(Q) \qquad \mathrm{pn}(\mu X(\tilde{x})\,.\,P) := \mathrm{pn}(P)$$

$$\mathrm{pn}((\boldsymbol{\nu} xy)P) := \mathrm{pn}(P) \qquad \mathrm{pn}(X\langle\tilde{x}\rangle) := \emptyset$$

Theorem 7 *(Fairness [54]). Suppose given a process $P \vdash \emptyset; \emptyset$. Then, for every $x \in \mathrm{pn}(P)$ there exists a process P' such that $P \longrightarrow^\star P'$ and $P' \xrightarrow{\alpha} Q$, for some process Q and label α with subject x.*

To illustrate APCP processes and their types, we give implementations of the three participants in G_{auth} in Section 1.

Example 1. Processes P, Q, and R are typed implementations for participants c, s, and a, respectively, where each process uses a single channel to perform the actions described by G_{auth}.

$$P := \mu X(c_\mu)\,.\,c_\mu \triangleright \begin{Bmatrix} \mathsf{login} : c_\mu(u)\,.\,\overline{c_\mu} \triangleleft \mathsf{passwd} \cdot \overline{c_\mu}[\boldsymbol{logmein345}] \cdot X\langle c_\mu\rangle, \\ \mathsf{quit} \;\; : c_\mu(w)\,.\,\overline{c_\mu} \triangleleft \mathsf{quit} \cdot \overline{c_\mu}[z] \cdot \mathbf{0} \end{Bmatrix}$$

$$\vdash c_\mu : \mu X\,.\,\&^2 \begin{Bmatrix} \mathsf{login} : \bullet \parr^3 \oplus^4 \{\mathsf{passwd} : \bullet \otimes^5 X\}, \\ \mathsf{quit} \;\; : \bullet \parr^3 \oplus^4 \{\mathsf{quit} : \bullet \otimes^5 \bullet\} \end{Bmatrix}$$

$$Q := \mu X(s_\mu)\,.\,\overline{s_\mu} \triangleleft \mathsf{login} \cdot \overline{s_\mu}[u] \cdot s_\mu \triangleright \{\mathsf{auth} : s_\mu(v)\,.\,X\langle s_\mu\rangle\}$$

$$\vdash s_\mu : \mu X\,.\,\oplus^0\{\mathsf{login} : \bullet \otimes^1 \&^{10}\{\mathsf{auth} : \bullet \parr^{11} X\}, \mathsf{quit} : \bullet \otimes^1 \bullet\}$$

$$R := \mu X(a_\mu)\,.\,a_\mu \triangleright \begin{Bmatrix} \mathsf{login} : a_\mu \triangleright \{\mathsf{passwd} : a_\mu(u)\,.\,\overline{a_\mu} \triangleleft \mathsf{auth} \cdot \overline{a_\mu}[v] \cdot X\langle a_\mu\rangle\}, \\ \mathsf{quit} \;\; : a_\mu \triangleright \{\mathsf{quit} : a_\mu(w)\,.\,\mathbf{0}\} \end{Bmatrix}$$

$$\vdash a_\mu : \mu X\,.\,\&^2 \begin{Bmatrix} \mathsf{login} : \&^6\{\mathsf{passwd} : \bullet \parr^7 \oplus^8 \{\mathsf{auth} : \bullet \otimes^9 X\}\}, \\ \mathsf{quit} \;\; : \&^6\{\mathsf{quit} : \bullet \parr^7 \bullet\} \end{Bmatrix}$$

Process P is a specific implementation for c, where we use '***logmein345***' to denote a closed channel endpoint representing the password string "logmein345". Similarly, Q is a specific implementation for s that continuously chooses the login branch.

Note that the processes above cannot be directly connected to each other to implement G_{auth}. Our goal is to enable the composition of (typed) implementations such as P, Q, and R in a correct and deadlock-free manner. We shall proceed as follows. After setting up the routers that enable the composition of these processes according to G_{auth} (Section 4), we will return to this example in Section 5. At that point, it will become clear that the priorities in the types of P, Q, and R were chosen to ensure the correct composition with their respective routers.

3. Global types and relative projection

We analyze multiparty protocols specified as *global types*. We consider a standard syntax, with session delegation and recursion, subsuming the one given in the seminal paper by Honda et al. [38]. In the following, we write $p, q, r, s, \ldots$ to denote *(protocol) participants*.

Definition 12 *(Types). Global types* G and *message types* S, T are defined as:

$$G ::= p \twoheadrightarrow q\{i\langle S\rangle\,.\,G\}_{i\in I} \mid \mu X\,.\,G \mid X \mid \bullet \mid \mathsf{skip}\,.\,G$$

$$S, T ::= !T\,.\,S \mid ?T\,.\,S \mid \oplus\{i : S\}_{i\in I} \mid \&\{i : S\}_{i\in I} \mid \bullet$$

We include basic types (e.g., unit, bool, int), which are all syntactic sugar for $\bullet$.

The type $p \twoheadrightarrow q\{i\langle S_i\rangle\,.\,G_i\}_{i\in I}$ specifies a direct exchange from participant p to participant q, which precedes protocol G_i: p chooses a label $i \in I$ and sends it to q along with a message of type S_i. Message exchange is *asynchronous*: the protocol can continue as G_i before the message has been received by q. The type $\mu X\,.\,G$ defines a recursive protocol: whenever a path of exchanges in G reaches the recursion variable X, the protocol continues as $\mu X\,.\,G$. The type $\bullet$ denotes the completed protocol. For technical convenience, we introduce the construct $\mathsf{skip}\,.\,G$, which denotes an unobservable step that precedes G.

Recursive definitions bind recursion variables, so recursion variables not bound by a recursive definition are free. We write $\mathrm{frv}(G)$ to denote the set of free recursion variables of G, and say G is *closed* if $\mathrm{frv}(G) = \emptyset$. Recursion in global types is tail-recursive and *contractive* (i.e. they contain no subexpressions of the form $\mu X_1 \ldots \mu X_n\,.\,X_1$). As for the session types in Section 2, we define the unfolding of a recursive global type by substituting copies of the recursive definition for recursive calls, i.e. $\mu X\,.\,G$ unfolds to $G\{\mu X\,.\,G/X\}$.

In approaches based on MPST, the grammar of global types specifies multiparty protocols but does not ensure their correct implementability; such guarantees are given in terms of *well-formedness*, defined as projectability onto all participants (cf. §3.2).

Message types S, T define binary protocols, not to be confused with the types in §2. Type $!T\,.\,S$ (resp. $?T\,.\,S$) denotes the output (resp. input) of a message of type T followed by the continuation S. Type $\oplus\{i : S_i\}_{i\in I}$ denotes *selection*: the output of choice for a label $i \in I$ followed by the continuation S_i. Type $\&\{i : S_i\}_{i\in I}$ denotes *branching*: the input of a label $i \in I$ followed by the continuation S_i. Type $\bullet$ denotes the end of the protocol. Note that, due to the tail-recursiveness of session and global types, there are no recursive message types.

It is useful to obtain the set of participants of a global type:

Definition 13 *(Participants).* We define the *set of participants* of global type G, denoted $\mathsf{prt}(G)$:

$$\mathsf{prt}(p \twoheadrightarrow q\{i\langle S_i\rangle \,.\, G_i\}_{i\in I}) := \{p,q\} \cup (\textstyle\bigcup_{i\in I} \mathsf{prt}(G_i)) \qquad \mathsf{prt}(\mathsf{skip}\,.\,G) := \mathsf{prt}(G) \qquad \mathsf{prt}(\bullet) := \emptyset$$
$$\mathsf{prt}(\mu X\,.\,G) := \mathsf{prt}(G) \qquad \mathsf{prt}(X) := \emptyset$$

3.1. Relative types

While a global type such as G_{auth} (1) describes a protocol from a vantage point, we introduce *relative types* that describe the interactions between *pairs* of participants. This way, relative types capture the peer-to-peer nature of multiparty protocols. We develop projection from global types onto relative types (cf. § 3.2) and use it to establish a new class of *well-formed* global types.

A choice between participants in a global type is *non-local* if it influences future exchanges between other participants. Our approach uses *dependencies* to expose these non-local choices in the relative types of these other participants.

Relative types express interactions between two participants. Because we obtain a relative type through projection of a global type, we know which participants are involved. Therefore, a relative type only mentions the sender of each exchange; we implicitly know that the recipient is the other participant.

Definition 14 *(Relative types). Relative types* R are defined as follows, where the S_i are message types (cf. Definition 12):

$$R ::= p\{i\langle S_i\rangle \,.\, R\}_{i\in I} \mid p?r\{i\,.\,R\}_{i\in I} \mid p!r\{i\,.\,R\}_{i\in I} \mid \mu X\,.\,R \mid X \mid \bullet \mid \mathsf{skip}\,.\,R$$

We detail the syntax above, given participants p and q.

- Type $p\{i\langle S_i\rangle \,.\, R_i\}_{i\in I}$ specifies that p must choose a label $i \in I$ and send it to q along with a message of type S_i after which the protocol continues with R_i.
- Given an r which is *not* involved in the relative type (i.e., $p \neq r, q \neq r$), type $p?r\{i\,.\,R_i\}_{i\in I}$ expresses a dependency: a non-local choice between p and r which influences the protocol between p and q. Here, the dependency indicates that after p receives from r the chosen label, p must forward it to q, determining the protocol between p and q.
- Similarly, type $p!r\{i\,.\,R_i\}_{i\in I}$ expresses a dependency, which indicates that after p sends to r the chosen label, p must forward it to q.
- Types $\mu X\,.\,R$ and X define recursion, just as their global counterparts.
- The type $\bullet$ specifies the end of the protocol between p and q.
- The type $\mathsf{skip}\,.\,R$ denotes an unobservable step that precedes R.

Definition 15 *(Participants of relative types).* We define the *set of participants* of relative type R, denoted $\mathsf{prt}(R)$:

$$\mathsf{prt}(p\{i\langle S_i\rangle \,.\, R_i\}_{i\in I}) := \{p\} \cup (\textstyle\bigcup_{i\in I} \mathsf{prt}(R_i)) \qquad \mathsf{prt}(\mathsf{skip}\,.\,R) := \mathsf{prt}(R) \qquad \mathsf{prt}(\bullet) := \emptyset$$
$$\mathsf{prt}(p?r\{i\,.\,R_i\}_{i\in I}) := \{p\} \cup (\textstyle\bigcup_{i\in I} \mathsf{prt}(R_i)) \qquad \mathsf{prt}(\mu X\,.\,R) := \mathsf{prt}(R) \qquad \mathsf{prt}(X) := \emptyset$$
$$\mathsf{prt}(p!r\{i\,.\,R_i\}_{i\in I}) := \{p\} \cup (\textstyle\bigcup_{i\in I} \mathsf{prt}(R_i))$$

We introduce some useful notation:

Notation 3.

- We write $p \twoheadrightarrow q : i\langle S\rangle \,.\, G$ for a global type with a single branch $p \twoheadrightarrow q\{i\langle S\rangle \,.\, G\}$ (and similarly for exchanges and dependencies in relative types).
- We omit unit message types from global and relative types, writing $i\,.\,G$ for $i\langle\mathsf{unit}\rangle\,.\,G$.
- Given $k > 1$, we write skip^k for a sequence of k skips.

3.2. Relative projection and well-formedness

We define *relative projection* for global types. We want relative projection to fail when it would return a non-contractive recursive type. To this end, we define a notion of contractiveness on relative types:

Definition 16 *(Contractive relative types).* Given a relative type R and a recursion variable X, we say R *is contractive on* X if either of the following holds:

- R contains an exchange, or

$$\mathsf{ddep}((p,q), s \twoheadrightarrow r\{i\langle S_i\rangle \,.\, G_i\}_{i\in I}) := \begin{cases} \mathsf{skip} \,.\, (G_{i'} \upharpoonright (p,q))\ [\text{any } i' \in I] & \text{if } \forall i, j. \\ & G_i \upharpoonright (p,q) = G_j \upharpoonright (p,q) \\ p!r\{i \,.\, (G_i \upharpoonright (p,q))\}_{i\in I} & \text{if } p = s \\ q!r\{i \,.\, (G_i \upharpoonright (p,q))\}_{i\in I} & \text{if } q = s \\ p?s\{i \,.\, (G_i \upharpoonright (p,q))\}_{i\in I} & \text{if } p = r \\ q?s\{i \,.\, (G_i \upharpoonright (p,q))\}_{i\in I} & \text{if } q = r \end{cases}$$

$$(s \twoheadrightarrow r\{i\langle S_i\rangle \,.\, G_i\}_{i\in I}) \upharpoonright (p,q) := \begin{cases} p\{i\langle S_i\rangle \,.\, (G_i \upharpoonright (p,q))\}_{i\in I} & \text{if } p = s \text{ and } q = r \\ q\{i\langle S_i\rangle \,.\, (G_i \upharpoonright (p,q))\}_{i\in I} & \text{if } q = s \text{ and } p = r \\ \mathsf{ddep}((p,q), s \twoheadrightarrow r\{i\langle S_i\rangle \,.\, G_i\}_{i\in I}) & \text{otherwise} \end{cases}$$

$$(\mu X \,.\, G) \upharpoonright (p,q) := \begin{cases} \mu X \,.\, (G \upharpoonright (p,q)) & \text{if } G \upharpoonright (p,q) \text{ defined and contractive on } X \\ \bullet & \text{otherwise} \end{cases}$$

$$X \upharpoonright (p,q) := X \qquad \bullet \upharpoonright (p,q) := \bullet \qquad (\mathsf{skip} \,.\, G) \upharpoonright (p,q) := \mathsf{skip} \,.\, (G \upharpoonright (p,q))$$

Above, skip^* denotes a sequence of zero or more skip.

Fig. 5. Dependency Detection (top), and Relative Projection (bottom, cf. Definition 17). When a side-condition does not hold, either is undefined.

- R ends in a recursive call on a variable other than X.

Relative projection then relies on the contractiveness of relative types. It also relies on an auxiliary function to determine if a dependency message is needed and possible.

Definition 17 *(Relative projection).* Given a global type G, we define its relative projection onto a pair of participants p and q, denoted $G \upharpoonright (p,q)$, by induction on the structure of G as given in Fig. 5 (bottom), using the auxiliary function ddep (cf. Fig. 5, top).

We discuss how Definition 17 projects global types onto a pair of participants (p,q), as per Fig. 5 (bottom). The most interesting case is the projection of a direct exchange $s \twoheadrightarrow r\{i\langle S_i\rangle \,.\, G_i\}_{i\in I}$. When the exchange involves both p and q, the projection yields an exchange between p and q with the appropriate sender. Otherwise, the projection relies on the function 'ddep' in Fig. 5 (top), which determines whether the exchange is a non-local choice for p and q and yields an appropriate projection accordingly:

- If the projections of all branches are equal, the exchange is not a non-local choice and ddep yields the unobservable step skip followed by the projection of any branch.
- If there are branches with different projections, the exchange is a non-local choice, so ddep yields a dependency if possible. If p or q is involved in the exchange, ddep yields an appropriate dependency (e.g., $p!r$ if p is the sender, or $q?s$ if q is the recipient). If neither p nor q are involved, then ddep cannot yield a dependency and projection is thus undefined.

The projection of $\mu X \,.\, G$' considers the projection of the body $G' \upharpoonright (p,q)$ to see whether p and q interact in G'. If $G' \upharpoonright (p,q)$ is a (possibly empty) sequence of skips followed by $\bullet$ or X, then p and q do not interact and the projection yields $\bullet$. Otherwise, p and q do interact and projection preserves the recursive definition. Note that Definition 16 (contractiveness) is key here: e.g., $G' \upharpoonright (p,q) = \mathsf{skip} \,.\, \mu Y \,.\, \mathsf{skip} \,.\, X$ is not contractive on X, so $(\mu X \,.\, G') \upharpoonright (p,q) = \bullet$. The projections of recursive calls X, of $\bullet$, and of skip are homomorphic.

Example 2 *(Projections of G_{auth}).* To demonstrate relative projection, let us consider again G_{auth}:

$$G_{\mathsf{auth}} = \mu X \,.\, s \twoheadrightarrow c \left\{ \begin{array}{l} \mathsf{login} \,.\, c \twoheadrightarrow a : \mathsf{passwd}\langle \mathsf{str}\rangle \,.\, a \twoheadrightarrow s : \mathsf{auth}\langle \mathsf{bool}\rangle \,.\, X, \\ \mathsf{quit} \,.\, c \twoheadrightarrow a : \mathsf{quit} \,.\, \bullet \end{array} \right\}$$

The relative projection onto (s,c) is straightforward, as there are no non-local choices to consider:

$$G_{\mathsf{auth}} \upharpoonright (s,c) = \mu X \,.\, s \left\{ \begin{array}{l} \mathsf{login} \,.\, \mathsf{skip}^2 \,.\, X, \\ \mathsf{quit} \,.\, \mathsf{skip} \,.\, \bullet \end{array} \right\}$$

However, compare the projection of the initial login branch onto (s,a) and (c,a) with the projection of the quit branch: they are different. Therefore, the initial exchange between s and c is a non-local choice in the protocols relative to (s,a) and (c,a). Since s is involved in this exchange, the non-local choice is detected by ddep:

$$\mathsf{ddep}((s,a), s \twoheadrightarrow c\{\mathsf{login}\ldots,\quad \mathsf{quit}\ldots\}) = s!c\{\mathsf{login}\ldots,\quad \mathsf{quit}\ldots\}$$

Hence, this non-local choice can be included in the relative projection onto (s,a) as a dependency:

$$G_{\mathsf{auth}} \upharpoonright (s,a) = \mu X\,.\, s!c \left\{ \begin{array}{l} \mathsf{login}\,.\,\mathsf{skip}\,.\,a : \mathsf{auth}\langle \mathsf{bool}\rangle\,.\,X, \\ \mathsf{quit}\,.\,\mathsf{skip}\,.\,\bullet \end{array} \right\}$$

Similarly, c is involved in the initial exchange, so the non-local choice can also be included in the relative projection onto (c,a) as a dependency:

$$G_{\mathsf{auth}} \upharpoonright (c,a) = \mu X\,.\, c?s \left\{ \begin{array}{l} \mathsf{login}\,.\,c : \mathsf{passwd}\langle \mathsf{str}\rangle\,.\,\mathsf{skip}\,.\,X, \\ \mathsf{quit}\,.\,c : \mathsf{quit}\langle \mathsf{unit}\rangle\,.\,\bullet \end{array} \right\}$$

Since relative types are relative to pairs of participants, the input order of participants for projection does not matter:

Proposition 8. *Suppose a global type G and distinct participants $p, q \in \mathsf{prt}(G)$.*

- *If $G \upharpoonright (p,q)$ is defined, then $G \upharpoonright (p,q) = G \upharpoonright (q,p)$ and $\mathsf{prt}(G \upharpoonright (p,q)) \subseteq \{p,q\}$;*
- *$G \upharpoonright (p,q)$ is undefined if and only if $G \upharpoonright (q,p)$ is undefined.*

Well-formed global types We may now define *well-formedness* for global types. Unlike usual MPST approaches, our definition relies exclusively on (relative) projection (Definition 17), and does not appeal to external notions such as merge and subtyping [39,58].

Definition 18 *(Relative well-formedness).* A global type G is *relative well-formed* if, for every distinct $p, q \in \mathsf{prt}(G)$, the projection $G \upharpoonright (p,q)$ is defined.

The following contrasts our new notion of relative well-formedness with notions of well-formedness based on the usual notion of local types [37,28].

Example 3. Consider the following global type involving participants p, q, r, s:

$$G_3 := p \twoheadrightarrow q \left\{ \begin{array}{l} 1\langle S_a\rangle\,.\,p \twoheadrightarrow r : 1\langle S_b\rangle\,.\,p \twoheadrightarrow s : 1\langle S_c\rangle\,.\,q \twoheadrightarrow r : 1\langle S_d\rangle\,.\,q \twoheadrightarrow s : 1\langle S_e\rangle\,.\,\bullet, \\ 2\langle S_f\rangle\,.\,r \twoheadrightarrow p : 2\langle S_g\rangle\,.\,s \twoheadrightarrow p : 2\langle S_h\rangle\,.\,r \twoheadrightarrow q : 2\langle S_i\rangle\,.\,s \twoheadrightarrow q : 2\langle S_j\rangle\,.\,\bullet \end{array} \right\}$$

The initial exchange between p and q is a non-local choice influencing the protocols between other pairs of participants. Well-formedness as in [37,28] forbids non-local choices. In contrast, G_3 is relative well-formed: p and q must both forward the selected label to both r and s. The dependencies in the following relative projections express precisely this:

$$G_3 \upharpoonright (p,r) = p!q\{1\,.\,p : 1\langle S_b\rangle\,.\,\mathsf{skip}^3\,.\,\bullet,\quad 2\,.\,r : 2\langle S_g\rangle\,.\,\mathsf{skip}^3\,.\,\bullet\}$$
$$G_3 \upharpoonright (p,s) = p!q\{1\,.\,\mathsf{skip}\,.\,p : 1\langle S_c\rangle\,.\,\mathsf{skip}^2\,.\,\bullet,\quad 2\,.\,\mathsf{skip}\,.\,s : 2\langle S_h\rangle\,.\,\mathsf{skip}^2\,.\,\bullet\}$$
$$G_3 \upharpoonright (q,r) = q?p\{1\,.\,\mathsf{skip}^2\,.\,q : 1\langle S_d\rangle\,.\,\mathsf{skip}\,.\,\bullet,\quad 2\,.\,\mathsf{skip}^2\,.\,r : 2\langle S_i\rangle\,.\,\mathsf{skip}\,.\,\bullet\}$$
$$G_3 \upharpoonright (q,s) = q?p\{1\,.\,\mathsf{skip}^3\,.\,q : 1\langle S_e\rangle\,.\,\bullet,\quad 2\,.\,\mathsf{skip}^3\,.\,s : 2\langle S_j\rangle\,.\,\bullet\}$$

Dependencies in relative types follow the non-local choices in the given global type: by implementing such choices, dependencies ensure correct projectability. They induce additional messages, but in our view this is an acceptable price to pay for an expressive notion of well-formedness based only on projection. It is easy to see that in a global type with n participants, the number of messages per communication is $\mathcal{O}(n)$—an upper-bound following from the worst-case scenario in which both sender and recipient have to forward a label to $n-2$ participants due to dependencies, as in the example above. However, in practice, sender and recipient will rarely both have to forward labels, let alone both to all participants.

4. Analyzing global types using routers

In this section, we develop our decentralized analysis of multiparty protocols (§3) using relative types (§3.1) and APCP (§2). The intended setup is as follows. Each participant's role in a global type G is implemented by a process, which is connected to a *router*: a process that orchestrates the participant's interactions in G. The resulting *routed implementations* (Definition 25) can then directly connect to each other to form a decentralized *network of routed implementations* that implements G. This way we realize the scenario sketched in Fig. 1 (left), which is featured in more detail in Fig. 6 (left).

Key in our analysis is the *synthesis* of a participant's router from a global type (§4.1). To assert well-typedness—and thus deadlock-freedom—of networks of routed implementations (Theorem 11), we extract binary session types from the global type and its associated relative types (§4.2):

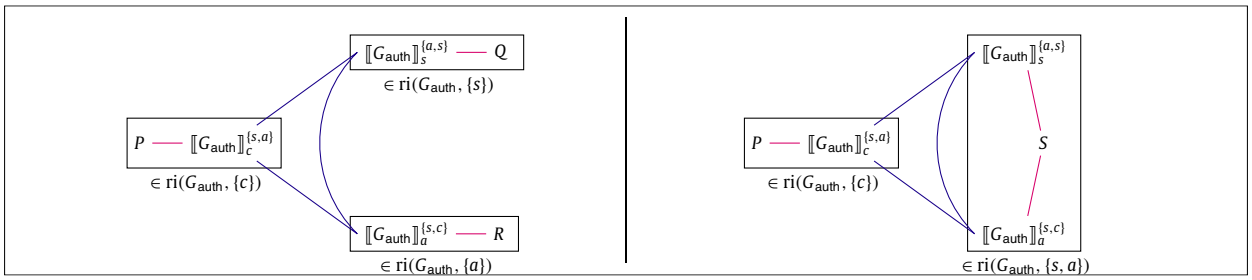

Fig. 6. Two different networks of routed implementations for G_{auth} (1), without interleaving (left) and with interleaving (right). For participants p and $\tilde{q}$, Definition 20 gives the router process $[\![G]\!]_p^{\tilde{q}}$ and Definition 25 gives the set $\text{ri}(G, \tilde{q})$. Lines indicate channels and boxes are local compositions of processes.

- from the global type we extract types for channels between implementations and routers;
- from the relative types we extract types for channels between pairs of routers.

After defining routers and showing their typability, we set up networks of routed implementations of global types (§ 4.3). To enable the transference of deadlock-freedom APCP to multiparty protocols, we then establish an operational correspondence between global types and networks of routed implementations (Theorems 19 and 23). Finally, to show that our routed approach strictly generalizes the prior centralized analyses [12,17], we define an orchestrated analysis of global types and show that it is behaviorally equivalent to a centralized composition of routers (§ 4.4).

In the following section (§ 5), we will show routers in action.

4.1. Synthesis of routers

We synthesize routers by decomposing each exchange in the global type into four sub-steps, which we motivate by considering the initial exchange from s to c in G_{auth} (1): $s \twoheadrightarrow c\{\text{login}\ldots,\quad \text{quit}\ldots\}$. As explained in Example 2, this exchange induces a dependency in the relative projections of G_{auth} onto (s, a) and (c, a). We decompose this initial exchange as follows, where P, Q, and R are the implementations of c, s, and a, respectively (given in Example 1) and $\mathcal{R}_x$ stands for the router of each $x \in \{s, c, a\}$. Below, multiple actions in one step happen concurrently:

1. Q sends $\ell \in \{\text{login}, \text{quit}\}$ to $\mathcal{R}_s$.
2. $\mathcal{R}_s$ sends ℓ to $\mathcal{R}_c$ (recipient) and $\mathcal{R}_a$ (output dependency). Q sends unit value v to $\mathcal{R}_s$.
3. $\mathcal{R}_c$ sends ℓ to P and $\mathcal{R}_a$ (input dependency). $\mathcal{R}_s$ forwards v to $\mathcal{R}_c$.
4. $\mathcal{R}_c$ forwards v to P. $\mathcal{R}_a$ sends ℓ to R.

In Section 4.2, we follow this decomposition to assign to each consecutive step a consecutive priority: this ensures the consistency of priority checks required to establish the deadlock-freedom of networks of routed implementations.

We define an algorithm that synthesizes a *router process* for a given global type and participant. More precisely: given G, a participant p, and $\tilde{q} = \text{prt}(G) \setminus \{p\}$, the algorithm generates a process, denoted $[\![G]\!]_p^{\tilde{q}}$, which connects with a process implementing p's role in G on channel μ_p; we shall write such channels in pink. This router for p connects with the routers of the other participants in G ($q_i \in \tilde{q}$) on channels $p_{q_1}, \ldots, p_{q_n}$; we shall write such channels in purple. (This convention explains the colors of the lines in Fig. 6.)

The router synthesis algorithm relies on relative projection to detect non-local choices; this way, the router can synchronize with the participant's implementation and with other routers appropriately. To this end, we define the predicate 'hdep', which is true for an exchange and a pair of participants if the exchange induces a dependency for either participant. Recall that relative projection produces a skip when an exchange is not non-local (cf. Fig. 5). Thus, hdep only holds true if relative projection does not produce a skip.

Definition 19. The predicate $\text{hdep}(q, p, G)$ is true if and only if

- $G = s \twoheadrightarrow r\{i\langle S_i\rangle\,.\,G_i\}_{i\in I}$ and $q \notin \{s, r\}$ and $p \in \{s, r\}$, and
- $\text{ddep}((p, q), G) \neq \text{skip}\,.\,R$ for all relative types R, where ddep is as in Fig. 5 (top).

Example 4. Consider the global type $G_{\text{h}} := p \twoheadrightarrow q\{\text{a}\,.\,p \twoheadrightarrow r : \text{a}\,.\,\bullet,\quad \text{b}\,.\,r \twoheadrightarrow p : \text{b}\,.\,\bullet\}$. We have that $\text{hdep}(q, p, G_{\text{h}})$ is false because the initial exchange in G_{h} is not a dependency for p and q, but $\text{hdep}(r, p, G_{\text{h}})$ is true because the initial exchange in G_{h} is indeed a dependency for p and r.

Algorithm 1: Synthesis of Router Processes (Definition 20).

1 **def** $[\![G]\!]_p^{\tilde{q}}$ **as**
2 **switch** G **do**
3 **case** $s \twoheadrightarrow r\{i\langle S_i\rangle . G_i\}_{i\in I}$ **do**
4 $\mathsf{deps} := \{q \in \tilde{q} \mid \mathsf{hdep}(q, p, G)\}$
5 **if** $p = s$ **then return** $\mu_p \triangleright \{i : \overline{p_r} \triangleleft i \cdot (\overline{p_q} \triangleleft i)_{q\in\mathsf{deps}} \cdot \mu_p(v) . \overline{p_r}[w] \cdot (v \leftrightarrow w \mid [\![G_i]\!]_p^{\tilde{q}})\}_{i\in I}$
6 **else if** $p = r$ **then return** $p_s \triangleright \{i : \overline{\mu_p} \triangleleft i \cdot (\overline{p_q} \triangleleft i)_{q\in\mathsf{deps}} \cdot p_s(v) . \overline{\mu_p}[w] \cdot (v \leftrightarrow w \mid [\![G_i]\!]_p^{\tilde{q}})\}_{i\in I}$
7 **else if** $p \notin \{s, r\}$ **then**
8 $\mathsf{depon}_s := (s \in \tilde{q} \wedge \mathsf{hdep}(p, s, G))$
9 $\mathsf{depon}_r := (r \in \tilde{q} \wedge \mathsf{hdep}(p, r, G))$
10 **if** depon_s *and* $\neg\mathsf{depon}_r$ **then return** $p_s \triangleright \{i : \overline{\mu_p} \triangleleft i \cdot [\![G_i]\!]_p^{\tilde{q}}\}_{i\in I}$
11 **else if** depon_r *and* $\neg\mathsf{depon}_s$ **then return** $p_r \triangleright \{i : \overline{\mu_p} \triangleleft i \cdot [\![G_i]\!]_p^{\tilde{q}}\}_{i\in I}$
12 **else if** depon_s *and* depon_r **then return** $p_s \triangleright \{i : \overline{\mu_p} \triangleleft i \cdot p_r \triangleleft \{i : [\![G_i]\!]_p^{\tilde{q}}\}\}_{i\in I}$
13 **else return** $[\![G_j]\!]_p^{\tilde{q}}$ for any $j \in I$
14 **case** $\mu X . G'$ **do**
15 $\tilde{q}' := \{q \in \tilde{q} \mid G \upharpoonright (p, q) \neq \bullet\}$
16 **if** $\tilde{q}' \neq \emptyset$ **then return** $\mu X(\mu_p, (p_q)_{q\in\tilde{q}'}) . [\![G']\!]_p^{\tilde{q}'}$
17 **else return 0**
18 **case** X **do return** $X\langle\mu_p, (p_q)_{q\in\tilde{q}}\rangle$
19 **case** $skip . G'$ **do return** $[\![G']\!]_p^{\tilde{q}}$
20 **case** $\bullet$ **do return 0**

Definition 20 *(Router synthesis).* Given a global type G, a participant p, and participants $\tilde{q}$, Algorithm 1 defines the synthesis of a *router process*, denoted $[\![G]\!]_p^{\tilde{q}}$, that interfaces the interactions of p with the other protocol participants according to G.

We often write $\mathcal{R}_p$ for $[\![G]\!]_p^{\mathsf{prt}(G)\setminus\{p\}}$ when G is clear from the context.

Algorithm 1 distinguishes six cases depending on the syntax of G (Definition 12). The key case is $s \twoheadrightarrow r\{i\langle U_i\rangle . G_i\}_{i\in I}$ (line 3). First, the algorithm computes a set deps of participants that depend on the exchange using hdep (cf. Definition 19). Then, the algorithm considers the three possibilities for p:

1. If $p = s$ then p is the sender (line 5): the algorithm returns a process that receives a label $i \in I$ over μ_p; sends i over p_r and over p_q for every $q \in \mathsf{deps}$; receives a channel v over μ_p; forwards v as w over p_r; and continues as $[\![G_i]\!]_p^{\tilde{q}}$.
2. If $p = r$ then p is the recipient (line 6): the algorithm returns a process that receives a label $i \in I$ over p_s; sends i over μ_p and over p_q for every $q \in \mathsf{deps}$; receives a channel v over p_s; forwards v as w over μ_p; and continues as $[\![G_i]\!]_p^{\tilde{q}}$.
3. Otherwise, if p is not involved (line 7), we use hdep to determine whether p depends on an output from s, an input from r, or on both (lines 8 and 9). If p only depends on the output from s, the algorithm returns a process that receives a label $i \in I$ over p_s; sends i over μ_p; and continues as $[\![G_i]\!]_p^{\tilde{q}}$ (line 10). If p only depends on an input from r, the returned process is similar; the only difference is that i is received over p_r (line 11).
   When p depends on *both* the output from s and on the input from r (line 12), the algorithm returns a process that receives a label $i \in I$ over p_s; sends i over μ_p; receives the label i over p_r; and continues as $[\![G_i]\!]_p^{\tilde{q}}$.
   If there are no dependencies, the returned process is $[\![G_j]\!]_p^{\tilde{q}}$, for arbitrary $j \in I$ (line 13).

In case $\mu X . G'$ (line 14), the algorithm stores in $\tilde{q}'$ those $q \in \tilde{q}$ that interact with p in G' (i.e. $\mu X . G' \upharpoonright (p, q) \neq \bullet$). Then, if $\tilde{q}'$ is non-empty (line 16), the algorithm returns a recursive definition with as context the channels p_q for $q \in \tilde{q}'$ and μ_p. Otherwise, the algorithm returns **0** (line 17). In case X (line 18), the algorithm returns a recursive call with as context the channels p_q for $q \in \tilde{q}$ and μ_p. In case skip . G' (line 19), it continues with G' immediately. Finally, in case $\bullet$ (line 20), the algorithm returns **0**.

Considering the number of steps required to return a process, the complexity of Algorithm 1 is linear in the size of the given global type (defined as the sum of the number of communications over all branches).

4.2. Types for the router's channels

Here, we obtain session types (cf. Definition 1) for (i) the channels between routers and implementations (§ 4.2.1) and for (ii) the channels between pairs of routers (§ 4.2.2). While the former are extracted from global types, the latter are extracted from relative types.

Below, $\mathsf{o} \in \mathbb{N}$ is arbitrary:

$$(\!|\bullet|\!) := \bullet \qquad (\!|!T\,.\,S|\!) := (\!|T|\!) \otimes^{\mathsf{o}} (\!|S|\!) \qquad (\!|\oplus\{i : S_i\}_{i\in I}|\!) := \oplus^{\mathsf{o}}\{i : (\!|S_i|\!)\}_{i\in I}$$

$$(\!|?T\,.\,S|\!) := (\!|T|\!) \parr^{\mathsf{o}} (\!|S|\!) \qquad (\!|\&\{i : S_i\}_{i\in I}|\!) := \&^{\mathsf{o}}\{i : (\!|S_i|\!)\}_{i\in I}$$

If $G = s \twoheadrightarrow r\{i\langle S_i\rangle\,.\,G_i\}_{i\in I}$,

$$G \downarrow^{\mathsf{o}} p := \begin{cases} \oplus^{\mathsf{o}}\{i : (\!|S_i|\!) \otimes^{\mathsf{o}+1} (G_i \downarrow^{\mathsf{o}+4} p)\}_{i\in I} & \text{if } p = s \\ \&^{\mathsf{o}+2}\{i : \overline{(\!|S_i|\!)} \parr^{\mathsf{o}+3} (G_i \downarrow^{\mathsf{o}+4} p)\}_{i\in I} & \text{if } p = r \\ \&^{\mathsf{o}+2}\{i : (G_i \downarrow^{\mathsf{o}+4} p)\}_{i\in I} & \text{if } p \notin \{s,r\} \text{ and } \mathrm{hdep}(p,s,G) \\ \&^{\mathsf{o}+3}\{i : (G_i \downarrow^{\mathsf{o}+4} p)\}_{i\in I} & \text{if } p \notin \{s,r\} \text{ and } \neg\mathrm{hdep}(p,s,G) \text{ and } \mathrm{hdep}(p,r,G) \\ G_{i'} \downarrow^{\mathsf{o}+4} p \ [\text{any } i' \in I] & \text{otherwise} \end{cases}$$

Otherwise,

$$\bullet \downarrow^{\mathsf{o}} p := \bullet \qquad (\mathsf{skip}\,.\,G') \downarrow^{\mathsf{o}} p := G' \downarrow^{\mathsf{o}+4} p \qquad X \downarrow^{\mathsf{o}} p := X$$

$$(\mu X\,.\,G') \downarrow^{\mathsf{o}} p := \begin{cases} \mu X\,.\,(G' \downarrow^{\mathsf{o}} p) & \text{if } G' \downarrow^{\mathsf{o}} p \text{ defined and contractive on } X \\ \bullet & \text{otherwise} \end{cases}$$

Fig. 7. Extracting Session Types from Message Types (top), and Local Projection: Extracting Session Types from a Global Type (bottom, cf. Definition 23).

4.2.1. The channels between routers and implementations

We begin with the session types for the channels between routers and implementations (given in pink), which we extract directly from the global type. A participant's implementation performs on this channel precisely those actions that the participant must perform as per the global type. Hence, we define this extraction as a form of *local projection* of the global type onto a *single participant*. The resulting session type may be used as a guidance for specifying a participant implementation, which can then connect to the router's dually typed channel endpoint.

Global types contain message types (Definition 12), so we must first define how we extract session types from message types. This is a straightforward definition, which leaves priorities unspecified: they do not matter for the typability of routers, which forward messages between implementations and other routers. Note that one must still specify these priorities when type-checking implementations, making sure they concur between sender and recipient.

Definition 21 *(From message types to session types).* We define the extraction of a session type from message type S, denoted $(\!|S|\!)$, by induction on the structure of S as in Fig. 7 (top).

We now define local projection. To deal with non-local choices, local projection incorporates dependencies by relying on the dependency detection of relative projection (cf. Definition 17). Also similar to relative projection, local projection relies on a notion of contractiveness for session types.

Definition 22 *(Contractive session types).* Given a session type A and a recursion variable X, we say A *is contractive on* X if either of the following holds:

- A contains a connective in $\{\otimes, \parr, \oplus, \&\}$, or
- A is a recursive call on a variable other than X.

Definition 23 *(Local projection: from global types to session types).* We define the local projection of global type G onto participant p with priority o, denoted $G \downarrow^{\mathsf{o}} p$, by induction on the structure of G as in Fig. 7 (bottom), relying on message type extraction (Definition 21) and the predicate hdep (Definition 19).

We consider the local projection of an exchange in a global type onto a participant p with priority o. The priorities in local projection reflect the four sub-steps into which we decompose exchanges in global types (cf. Section 4.1). There are three possibilities, depending on the involvement of p in the exchange:

1. If p is the sender, local projection specifies a choice ($\oplus$) between the exchange's labels at priority o and an output ($\otimes$) of the associated message type at priority $\mathsf{o}+1$, followed by the projection of the chosen branch at priority $\mathsf{o}+4$.
2. If p is the recipient, local projection specifies a branch (&) on the exchange's labels at priority $\mathsf{o}+2$ and an input ($\parr$) of the associated message type at priority $\mathsf{o}+3$, followed by the projection of the chosen branch at priority $\mathsf{o}+4$.
3. If p is neither sender nor recipient, local projection uses the predicate hdep (Definition 19) to detect a dependency on the sender's output or the recipient's input. If there is a dependency on the output, local projection specifies a branch

$$\langle\!| s\{i\langle S_i\rangle \,.\, R_i\}_{i\in I} |\!\rangle^{\mathsf{o}}_{p\rangle q} := \begin{cases} \oplus^{\mathsf{o}+1}\left\{i : \langle\!| S_i |\!\rangle \otimes^{\mathsf{o}+2} \langle\!| R_i |\!\rangle^{\mathsf{o}+4}_{p\rangle q}\right\}_{i\in I} & \text{if } p = s \\ \&^{\mathsf{o}+1}\left\{i : \overline{\langle\!| S_i |\!\rangle} \parr^{\mathsf{o}+2} \langle\!| R_i |\!\rangle^{\mathsf{o}+4}_{p\rangle q}\right\}_{i\in I} & \text{if } q = s \end{cases}$$

$$\langle\!| r?s\{i \,.\, R_i\}_{i\in I} |\!\rangle^{\mathsf{o}}_{p\rangle q} := \begin{cases} \oplus^{\mathsf{o}+2}\left\{i : \langle\!| R_i |\!\rangle^{\mathsf{o}+4}_{p\rangle q}\right\}_{i\in I} & \text{if } p = r \\ \&^{\mathsf{o}+2}\left\{i : \langle\!| R_i |\!\rangle^{\mathsf{o}+4}_{p\rangle q}\right\}_{i\in I} & \text{if } q = r \end{cases}$$

$$\langle\!| s!r\{i \,.\, R_i\}_{i\in I} |\!\rangle^{\mathsf{o}}_{p\rangle q} := \begin{cases} \oplus^{\mathsf{o}+1}\left\{i : \langle\!| R_i |\!\rangle^{\mathsf{o}+4}_{p\rangle q}\right\}_{i\in I} & \text{if } p = s \\ \&^{\mathsf{o}+1}\left\{i : \langle\!| R_i |\!\rangle^{\mathsf{o}+4}_{p\rangle q}\right\}_{i\in I} & \text{if } q = s \end{cases}$$

$$\langle\!| \bullet |\!\rangle^{\mathsf{o}}_{p\rangle q} := \bullet \qquad \langle\!| \mathsf{skip} \,.\, R |\!\rangle^{\mathsf{o}}_{p\rangle q} := \langle\!| R |\!\rangle^{\mathsf{o}+4}_{p\rangle q} \qquad \langle\!| \mu X \,.\, R |\!\rangle^{\mathsf{o}}_{p\rangle q} := \mu X \,.\, \langle\!| R |\!\rangle^{\mathsf{o}}_{p\rangle q} \qquad \langle\!| X |\!\rangle^{\mathsf{o}}_{p\rangle q} := X$$

Fig. 8. Extracting Session Types from Relative Types (cf. Definition 24).

on the exchange's labels at priority $\mathsf{o}+2$. If there is a dependency on the input, local projection specifies a branch at priority $\mathsf{o}+3$. Otherwise, when there is no dependency at all, local projection simply continues with the projection of any branch at priority $\mathsf{o}+4$.

Projection only preserves recursive definitions if they contain actual behavior (i.e. the projection of the recursive loop is contractive, cf. Definition 22). The projections of $\bullet$ and recursion variables are homomorphic. The projection of skip simply projects the skip's continuation, at priority $\mathsf{o}+4$ to keep the priority aligned with the priorities of the other types of the router.

4.2.2. The channels between pairs of routers

For the channels between pairs of routers (given in purple), we extract session types from relative types (Definition 14). Considering a relative type that describes the protocol between p and q, this entails decomposing it into a type for p and a dual type for q.

Definition 24 *(From relative types to session types).* We define the extraction of a session type from relative type R between p and q at p's perspective with priority o, denoted $\langle\!| R |\!\rangle^{\mathsf{o}}_{p\rangle q}$, by induction on the structure of R as in Fig. 8.

Here, extraction is *directional*: in $\langle\!| R |\!\rangle^{\mathsf{o}}_{p\rangle q}$, the annotation $p\rangle q$ says that the session type describes the perspective of p's router with respect to q's. Messages with sender p are decomposed into selection ($\oplus$) at priority $\mathsf{o}+1$ followed by output ($\otimes$) at priority $\mathsf{o}+2$. Dependencies on messages received by p become selection types ($\oplus$) at priority $\mathsf{o}+1$, and dependencies on messages sent by p become selection types ($\oplus$) at priority $\mathsf{o}+2$. Messages from q and dependencies on q yield dual types. Extraction from $\bullet$ and recursion is homomorphic, and extraction from skip simply extracts from the skip's continuation at priority $\mathsf{o}+4$.

This way, the channel endpoint of p's router that connects to q's router will be typed $\langle\!| G \upharpoonright (p,q) |\!\rangle^{\mathsf{o}}_{p\rangle q}$, i.e. the session type extracted from the relative projection of G onto p, q at p's perspective. Similarly, the endpoint of this channel at q's router will have the type $\langle\!| G \upharpoonright (p,q) |\!\rangle^{\mathsf{o}}_{q\rangle p}$, i.e. the same relative projection but at q's perspective. Clearly, these session types must be dual.

Theorem 9. *Given a relative well-formed global type G and $p, q \in \mathsf{prt}(G)$,*

$$\langle\!| G \upharpoonright (p,q) |\!\rangle^{\mathsf{o}}_{p\rangle q} = \overline{\langle\!| G \upharpoonright (p,q) |\!\rangle^{\mathsf{o}}_{q\rangle p}}.$$

Proof. By construction from Definition 17 and Definition 24. □

4.3. Networks of routed implementations

Having defined routers and types for their channels, we now turn to defining *networks of routed implementations*, i.e., process networks of routers and implementations that correctly represent a given multiparty protocol. Then, we appeal to the types obtained in § 4.2 to establish the typability of routers (Theorem 11). Finally, we show that all networks of routed implementations of well-formed global types are deadlock-free (Theorem 18), and that networks of routed implementations behave as depicted by the global types from which they are generated (Theorems 19 and 23).

We begin by defining routed implementations, which connect implementations of subsets of protocol participants with routers:

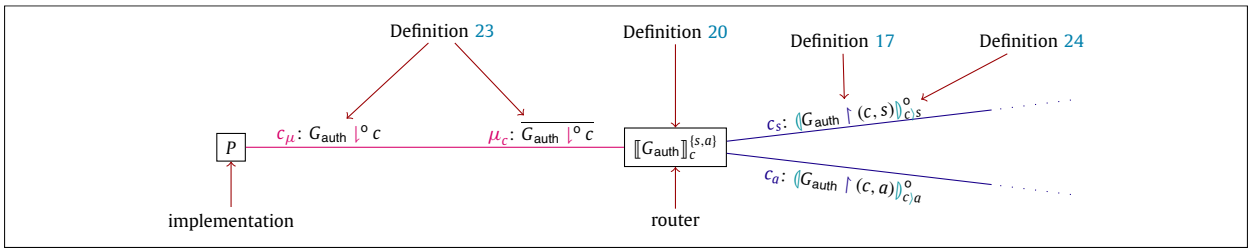

Fig. 9. Overview of Theorem 11, with the definitions and notations for synthesizing and typing routers, using participant c of G_{auth} implemented as P (cf. Example 1). Boxes indicate processes and lines indicate channels.

Definition 25 *(Routed implementations).* Given a closed, relative well-formed global type G, for participants $\tilde{p} \subseteq \mathsf{prt}(G)$, the *set of routed implementations* of $\tilde{p}$ in G is defined as follows (cf. Definition 23 for local projection '$\upharpoonright$' and Definition 20 for router synthesis '$[\![\ldots]\!]$'):

$$\mathrm{ri}(G,\tilde{p}) := \left\{ (\boldsymbol{\nu}\, \mu_p p_\mu)_{p\in\tilde{p}}\, (Q \mid \textstyle\prod_{p\in\tilde{p}} \mathcal{R}_p) \;\middle|\; \begin{array}{l} Q \vdash \emptyset;\Gamma, (p_\mu : G \upharpoonright^0 p)_{p\in\tilde{p}} \\ \wedge\, \forall p \in \tilde{p}.\, \mathcal{R}_p = [\![G]\!]_p^{\mathsf{prt}(G)\setminus\{p\}} \end{array} \right\}$$

We write $\mathcal{N}_{\tilde{p}}, \mathcal{N}'_{\tilde{p}}, \ldots$ to denote elements of $\mathrm{ri}(G,\tilde{p})$.

Thus, the composition of a collection of routers and an implementation Q is a routed implementation as long as Q can be typed in a context that includes the corresponding projected types. Note that the parameter $\tilde{p}$ indicates the presence of *interleaving*: when $\tilde{p}$ is a singleton, the set $\mathrm{ri}(G,\tilde{p})$ contains processes in which there is a single router and the implementation Q is single-threaded (non-interleaved); more interestingly, when $\tilde{p}$ includes two or more participants, the set $\mathrm{ri}(G,\tilde{p})$ consists of processes in which the implementation Q interleaves the roles of the multiple participants in $\tilde{p}$.

A network of routed implementations of a global type, or simply a *network*, is then the composition of any combination of routed implementations that together account for all the protocol's participants. Hence, we define sets of networks, quantified over all possible combinations of sets of participants and their respective routed implementations. The definition relies on *complete partitions* of the participants of a global type, i.e., a split of $\mathsf{prt}(G)$ into non-empty, disjoint subsets whose union yields $\mathsf{prt}(G)$.

Definition 26 *(Networks).* Suppose given a closed, relative well-formed global type G. Let $\mathbb{P}_G$ be the set of all complete partitions of $\mathsf{prt}(G)$ with elements $\pi, \pi', \ldots$. The *set of networks* of G is defined as

$$\mathrm{net}(G) := \left\{ (\boldsymbol{\nu}\, p_q q_p)_{p,q\in\mathsf{prt}(G)} (\textstyle\prod_{\tilde{p}\in\pi} \mathcal{N}_{\tilde{p}}) \;\middle|\; \pi \in \mathbb{P}_G \wedge \forall \tilde{p} \in \pi.\, \mathcal{N}_{\tilde{p}} \in \mathrm{ri}(G,\tilde{p}) \right\}.$$

We write $\mathcal{N}, \mathcal{N}', \ldots$ to denote elements of $\mathrm{net}(G)$.

Example 5. Fig. 6 depicts two networks in $\mathrm{net}(G_{\mathsf{auth}})$ related to different partitions of $\mathsf{prt}(G_{\mathsf{auth}})$, namely $\{\{a\},\{s\},\{c\}\}$ (non-interleaved) on the left and $\{\{a,s\},\{c\}\}$ (interleaved) on the right.

Because a network $\mathcal{N}$ may not be typable under the empty typing context, we have the following definition to "complete" networks.

Definition 27 *(Completable networks).* Suppose given a network $\mathcal{N}$ such that $\mathcal{N} \vdash \emptyset;\Gamma$. We say that $\mathcal{N}$ is *completable* if (i) Γ is empty or (ii) there exist $\tilde{v}, \tilde{w}$ such that $(\boldsymbol{\nu}\, \tilde{v}\tilde{w})\mathcal{N} \vdash \emptyset;\emptyset$. When $\mathcal{N}$ is completable, we write $\mathcal{N}^{\circlearrowleft}$ to stand for $\mathcal{N}$ (if $\mathcal{N} \vdash \emptyset;\emptyset$) or $(\boldsymbol{\nu}\, \tilde{v}\tilde{w})\mathcal{N}$ (otherwise).

Proposition 10. *For any closed, relative well-formed global type G, there exists at least one completable network $\mathcal{N} \in \mathrm{net}(G)$.*

Proof. To construct a completable network in $\mathrm{net}(G)$, we construct a routed implementation (Definition 25) for every $p \in \mathsf{prt}(G)$. Given a $p \in \mathsf{prt}(G)$, by Proposition 1, there exists $Q \vdash \emptyset;\, p_\mu : G \upharpoonright^0 p$. Composing each such characteristic implementation process with routers, and then composing the routed implementations, we obtain a network $\mathcal{N} \in \mathrm{net}(G)$, where $\mathcal{N} \vdash \emptyset;\emptyset$. Hence, $\mathcal{N}$ is completable. □

4.3.1. The typability of routers

We wish to establish that the networks of a global type are deadlock-free. This result, formalized by Theorem 18 (Page 31), hinges on the typability of routers, which we address next. Fig. 9 gives an overview of the definitions and notations involved in this theorem's statement.

Theorem 11. *Suppose given a closed, relative well-formed global type G, and a $p \in \mathsf{prt}(G)$. Then,*

$$[\![G]\!]_p^{\mathsf{prt}(G)\setminus\{p\}} \vdash \emptyset;\ \mu_p : \overline{G \downarrow^0 p},\ \left(p_q : (\!|G \upharpoonright (p,q)|\!)^0_{p,q}\right)_{q\in\mathsf{prt}(G)\setminus\{p\}}.$$

This result is a corollary of Theorem 16 (Page 23), which we show next. We give a full proof on Page 30, after the proof of Theorem 16.

Alarm processes We focus on networks of routed implementations—compositions of synthesized routers and well-typed processes. However, in order to establish the typability of routers we must account for an edge case that goes beyond these assumptions, namely when a routed implementation is connected to some undesirable implementation, not synthesized by Algorithm 1. Consider the following example:

Example 6. Consider again the global type G_{auth}, which, for the purpose of this example, we write as follows:

$$G_{\mathsf{auth}} = s \twoheadrightarrow c \begin{Bmatrix} \mathsf{login} : G_{\mathsf{login}}, \\ \mathsf{quit} : G_{\mathsf{quit}} \end{Bmatrix}$$

As established in Example 2, the initial exchange between s and c determines a dependency for the interactions of a with both s and c. Therefore, the implementation of a needs to receive the choice between login and quit from the implementations of both s and c. An undesirable implementation for c, without a router, could be for instance as follows:

$$R' := c_s \triangleright \begin{Bmatrix} \mathsf{login} : c_a \triangleleft \mathsf{quit} \cdot \ldots, \\ \mathsf{quit} : c_a \triangleleft \mathsf{quit} \cdot \ldots \end{Bmatrix}$$

Notice how R' always sends to a the label quit, even if the choice made by s (and sent to c) is login. Now, if s chooses login, the router of a is in limbo: on the one hand, it expects s to behave as specified in G_{login}; on the other hand, it expects c to behave as specified in G_{quit}. Clearly, the router of a is in an inconsistent state due to c's implementation.

Because routers always forward the chosen label correctly, this kind of undesirable behavior never occurs in the networks of Definition 26—we state this formally in § 4.3.2 (Theorem 17). Still, in order to prove that our routers are well-typed, we must accommodate the possibility that a router ends up in an undesirable state due to inconsistent forwarding. For this, we extend APCP with an *alarm process* that signals an inconsistency on a given set of channel endpoints.

Definition 28 *(Alarm process).* Given channel endpoints $\tilde{x} = x_1, \ldots, x_n$, we write $\mathsf{alarm}(\tilde{x})$ to denote an inconsistent state on those endpoints.

In a way, $\mathsf{alarm}(\tilde{x})$ is closer to an observable action (a "barb") than to an actual process term: $\mathsf{alarm}(\tilde{x})$ does not have reductions, and no process from Fig. 3 (top) can reduce to $\mathsf{alarm}(\tilde{x})$. We assume that $\mathsf{alarm}(\tilde{x})$ does not occur in participant implementations (cf. Q in Definition 25); we treat it as a process solely for the purpose of refining the router synthesis algorithm (Algorithm 1) with the possibility of inconsistent forwarding. The refinement concerns the process on line 12:

$$p_s \triangleright \left\{ i : \overline{\mu_p} \triangleleft i \cdot p_r \triangleleft \{ i : [\![G_i]\!]_p^{\tilde{q}} \} \right\}_{i\in I}$$

We extend it with additional branches, as follows:

$$p_s \triangleright \left\{ i : \overline{\mu_p} \triangleleft i \cdot p_r \triangleleft \left(\begin{array}{l} \{ i : [\![G_i]\!]_p^{\tilde{q}} \} \\ \underline{\cup\, \{ i' : \mathsf{alarm}(\mu_p, (p_q)_{q\in\tilde{q}}) \}_{i'\in I\setminus\{i\}}} \end{array} \right) \right\}_{i\in I} \tag{2}$$

This new process for line 12 captures the kind of inconsistency illustrated by Example 6, which occurs when a label $i \in I$ is received over p_s after which a label $i' \in I \setminus \{i\}$ is received over p_r. We account for this case by using the underlined alarm processes.

Routers are then made of processes as in Fig. 3 (top), selectively extended with alarms as just described. Because $\mathsf{alarm}(\tilde{x})$ merely acts as an observable that signals undesirable behavior, we find it convenient to type it using the following axiom:

$$\frac{}{\mathsf{alarm}(x_1, \ldots, x_n) \vdash \Omega; x_1 : A_1, \ldots, x_n : A_n}\ \textsc{Alarm}$$

where the recursive context Ω and types $A_1, \ldots, A_n$ are arbitrary.

Context-based typability Considering the refinement of Algorithm 1 with alarm processes, we prove Theorem 16 on Page 23, from which Theorem 11 follows as a corollary. It relies on some additional auxiliary definitions and results.

To type the router for a participant at any point in the protocol, we need the definition of the entire protocol. It is not enough to only consider the current (partial) protocol at such points: we need information about bound recursion variables in order to perform unfolding in types. To this end, we define *global contexts*, that allow us to look at part of a protocol while retaining definitions that concern the entire protocol.

Definition 29 *(Global contexts). Global contexts* C are given by the following grammar:

$$C ::= p \twoheadrightarrow q \begin{pmatrix} \{i\langle S\rangle \,.\, G\}_{i\in I} \\ \cup \{i'\langle S\rangle \,.\, C\}_{i'\notin I} \end{pmatrix} \mid \mathsf{skip} \,.\, C \mid \mu X \,.\, C \mid []$$

We often simply write "context" when it is clear that we are referring to a global context. Given a context C and a global type G, we write $C[G]$ to denote the global type obtained by replacing the hole [] in C with G. If $G = C[G_s]$ for some context C and global type G_s, then we write $G_s \preceq_C G$.

As mentioned before, a context captures information about the recursion variables that are bound at any given point in a global type. Our goal is to obtain a *context-based* typability result for routers.

The order in which recursive variables are bound is important to correctly unfold types:

Example 7. Consider the following global type with three nested recursive definitions:

$$G_{\mathsf{rec}} = \mu X \,.\, a \twoheadrightarrow b : 1 \,.\, \mu Y \,.\, a \twoheadrightarrow b : 2 \,.\, \mu Z \,.\, a \twoheadrightarrow b\{\mathsf{x} : X, \quad \mathsf{y} : Y, \quad \mathsf{z} : Z\}$$

To type the router for, e.g., a at the final exchange between a and b, we need to be aware of the unfolding of recursion. The recursion on X, Y, and Z have all to be unfolded, and the recursion on Z must include first the unfolding of X and then the unfolding of Y, which must in turn include the prior unfolding of X.

To account for nested recursions, the following definition gives the bound variables of a context exactly in the order in which they appear:

Definition 30 *(Recursion binders of contexts).* Given a global context C, the *sequence of recursion binders to the hole* of C, denoted $\mathrm{ctxbind}(C)$, is defined as follows:

$$\mathrm{ctxbind}(\mu X \,.\, C) := (X, \mathrm{ctxbind}(C)) \qquad \mathrm{ctxbind}(\mathsf{skip} \,.\, C) := \mathrm{ctxbind}(C) \qquad \mathrm{ctxbind}([]) := ()$$

$$\mathrm{ctxbind}(p \twoheadrightarrow q \begin{pmatrix} \{i\langle S_i\rangle \,.\, G_i\}_{i\in I} \\ \cup \{i'\langle S_{i'}\rangle \,.\, C\}_{i'\notin I} \end{pmatrix}) := \mathrm{ctxbind}(C)$$

Given $G_s \preceq_C G$, the sequence of recursion binders of G_s, denoted $\mathrm{subbind}(G_s, G)$, is defined as $\mathrm{ctxbind}(C)$.

The following retrieves the body of a recursive definition from a global context, informing us on how to unfold types:

Definition 31 *(Recursion extraction).* The function $\mathrm{recdef}(X, G)$ extracts the recursive definition on X from G, i.e. $\mathrm{recdef}(X,G) = G'$ if $\mu X \,.\, G' \preceq_C G$ for some context C. Also, $\mathrm{recCtx}(X, G)$ extracts the context of the recursive definition on X in G, i.e. $\mathrm{recCtx}(X, G) = C$ if $\mu X \,.\, \mathrm{recdef}(X, G) \preceq_C G$.

When unfolding bound recursion variables, we need the priorities of the unfolded types. The following definition gives a priority that is expected at the hole in a context, as well as the priority expected at any recursive definition in a global type:

Definition 32 *(Absolute priorities of contexts).* Given a context C and $\mathsf{o} \in \mathbb{N}$, we define $\mathrm{ctxpri}^{\mathsf{o}}(C)$ as follows:

$$\mathrm{ctxpri}^{\mathsf{o}}([]) := \mathsf{o} \qquad \mathrm{ctxpri}^{\mathsf{o}}(\mathsf{skip} \,.\, C) := \mathrm{ctxpri}^{\mathsf{o}+4}(C) \qquad \mathrm{ctxpri}^{\mathsf{o}}(\mu X \,.\, C) := \mathrm{ctxpri}^{\mathsf{o}}(C)$$

$$\mathrm{ctxpri}^{\mathsf{o}}(p \twoheadrightarrow q \begin{pmatrix} \{i\langle S_i\rangle \,.\, G_i\}_{i\in I} \\ \cup \{i'\langle S_{i'} \,.\, C\}_{i'\notin I} \end{pmatrix}) := \mathrm{ctxpri}^{\mathsf{o}+4}(C)$$

Then, the *absolute priority* of C, denoted $\mathrm{ctxpri}(C)$, is defined as $\mathrm{ctxpri}^0(C)$. The absolute priority of X in G, denoted $\mathrm{varpri}(X, G)$, is defined as $\mathrm{ctxpri}(C)$ for some context C such that $\mu X \,.\, \mathrm{recdef}(X, G) \preceq_C G$.

To avoid non-contractive recursive types, relative projection (cf. Fig. 5) closes a type when the participants do not interact inside a recursive definition. Hence, when typing a router for a recursive definition, we must determine which pairs of participants are "active" at any given point in a protocol, and close the connections with the "inactive" participants.

Example 8. Consider the following global type, where a client (c) requests two independent, infinite Fibonacci sequences (f_1 and f_2):

$$G_{\mathsf{fib}} = c \twoheadrightarrow f_1 : \mathsf{init}\langle \mathsf{int} \times \mathsf{int} \rangle \,.\, c \twoheadrightarrow f_2 : \mathsf{init}\langle \mathsf{int} \times \mathsf{int} \rangle \,.\, \underbrace{\mu X \,.\, f_1 \twoheadrightarrow c : \mathsf{next}\langle \mathsf{int} \rangle \,.\, f_2 \twoheadrightarrow c : \mathsf{next}\langle \mathsf{int} \rangle \,.\, X}_{G'_{\mathsf{fib}}}$$

Participants f_1 and f_2 do not interact with each other in the body of the recursion, as formalized by their relative projection:

$$\mathsf{recdef}(X, G_{\mathsf{fib}}) \upharpoonright (f_1, f_2) = \mathsf{skip} \,.\, \mathsf{skip} \,.\, X$$

Hence, $G'_{\mathsf{fib}} \upharpoonright (f_1, f_2) = \bullet$, and f_1 and f_2 do not form an active pair of participants for the recursion in G_{fib}. Therefore, f_1's router closes its connection with f_2's router at the start of the recursion on X, and vice versa.

The following definition uses relative projection to determine the pairs of active participants at the hole of a context, as well as at any recursive definition in a global type. We consider pairs of participants (p, q) and (q, p) to be equivalent.

Definition 33 *(Active participants).* Suppose given a relative well-formed global type G. The following mutually defined functions compute sets of *pairs of active participants* for recursive definitions and contexts, denoted $\mathsf{recactive}(X, G)$ and $\mathsf{active}(C, G)$, respectively.

$$\mathsf{recactive}(X, G) := \{(p, q) \in \mathsf{active}(\mathsf{recCtx}(X, G), G) \mid (\mu X \,.\, \mathsf{recdef}(X, G)) \upharpoonright (p, q) \neq \bullet\}$$

$$\mathsf{active}(C, G) := \begin{cases} \mathsf{recactive}(Y, G) & \text{if } \mathsf{ctxbind}(C) = (\tilde{X}, Y) \\ \mathsf{prt}(G)^2 & \text{otherwise} \end{cases}$$

The interdependency between $\mathsf{recactive}(X, G)$ and $\mathsf{active}(C, G)$ is well-defined: the former function considers the active participants of a context, which contains less recursive definitions.

When typing a router for a given protocol, we have to keep track of assignments in the recursive context at any point in the protocol. The following two lemmas ensure that the active participants of recursive definitions are consistent with the active participants of their bodies.

Lemma 12. *Suppose given a closed, relative well-formed global type G, and a global type G_s and context C such that $G_s \leq_C G$. For any $Z \in \mathsf{ctxbind}(C)$, $\mathsf{active}(C, G) \subseteq \mathsf{recactive}(Z, G)$.*

Proof. Take any $Z \in \mathsf{ctxbind}(C)$. Then $\mathsf{ctxbind}(C) = (\tilde{X}, Y)$. By definition, $\mathsf{active}(C, G) = \mathsf{recactive}(Y, G)$. If $Y = Z$, the thesis is proven. Otherwise, by definition, $\mathsf{recactive}(Y, G) \subseteq \mathsf{active}(\mathsf{recCtx}(Y, G), G)$. Since the recursive definition on Z appears in $\mathsf{recCtx}(Y, G)$, it follows by induction on the size of $\tilde{X}$ that $\mathsf{active}(\mathsf{recCtx}(Y, G), G) \subseteq \mathsf{recactive}(Z, G)$. This proves the thesis. □

The following lemma ensures that when typing a recursive call, the endpoints given as context for the recursive call concur with the endpoints in the recursive context:

Lemma 13. *Suppose given a closed, relative well-formed global type G, a recursion variable Z, and a context C such that $Z \leq_C G$. Then, $\mathsf{active}(C, G) = \mathsf{recactive}(Z, G)$.*

Proof. Because $G = C[Z]$ and G is closed (i.e. $\mathsf{frv}(G) = \emptyset$), there is a recursive definition on Z in G. Hence, $\mathsf{ctxbind}(C) \neq \emptyset$, i.e. $\mathsf{ctxbind}(C) = (\tilde{X}, Y)$ and $\mathsf{active}(C, G) = \mathsf{recactive}(Y, G)$. If $Y = Z$, the thesis is proven. Otherwise, the recursive definition on Y in G appears somewhere inside the recursive definition on Z. Suppose, for contradiction, that $\mathsf{active}(C, G) \neq \mathsf{recactive}(Z, G)$. There are two cases: there exists $(p, q) \in \mathsf{prt}(G)^2$ s.t. (i) $(p, q) \in \mathsf{active}(C, G)$ and $(p, q) \notin \mathsf{recactive}(Z, G)$, or (ii) $(p, q) \in \mathsf{recactive}(Z, G)$ and $(p, q) \notin \mathsf{active}(C, G)$. Case (i) contradicts Lemma 12.

In case (ii), $(\mu Z \,.\, \mathsf{recdef}(Z, G)) \upharpoonright (p, q) \neq \bullet$ and $(\mu Y \,.\, \mathsf{recdef}(Y, G)) \upharpoonright (p, q) = \bullet$. The recursive call on Z in G appears somewhere inside the recursive definition on Y, and hence $\mathsf{recdef}(Y, G) \upharpoonright (p, q)$ contains the recursive call on Z. This means that $\mathsf{recdef}(Y, G) \upharpoonright (p, q)$ is contractive on Y (Definition 16), and hence $(\mu Y \,.\, \mathsf{recdef}(Y, G)) \upharpoonright (p, q) \neq \bullet$, contradicting the assumption. □

Our typability result for routers relies on relative and local projection. Hence, we need to guarantee that all the projections we need at any given point of a protocol are defined. The following result shows a form of compositionality for relative and local projection, guaranteeing the definedness of projections for all active participants of a given context:

Proposition 14. *Suppose given a closed, relative well-formed global type G, and a global type G_s such that $G_s \preceq_C G$. Then, for every $(p,q) \in \mathsf{active}(C,G)$, the relative projection $G_s \upharpoonright (p,q)$ is defined. Also, for every $p \in \{p \in \mathsf{prt}(G) \mid \exists q \in \mathsf{prt}(G).\ (p,q) \in \mathsf{active}(C,G)\}$, the local projection $G_s \downarrow^{\mathsf{o}} p$ is defined for any priority o.*

Proof. Suppose that, for contradiction, $G_s \upharpoonright (p,q)$ is undefined. We show by induction on the structure of C that this means that $G \upharpoonright (p,q)$ is undefined, contradicting the relative well-formedness of G.

- Hole: $C = []$. We have $G_s = G$, and the thesis follows immediately.
- Exchange: $C = r \twoheadrightarrow s \begin{pmatrix} \{i\langle S_i\rangle\ .\ G_i\}_{i\in I} \\ \cup\ \{i'\langle S_{i'}\rangle\ .\ C'\}_{i'\notin I} \end{pmatrix}$. By the IH, $C'[G_s] \upharpoonright (p,q)$ is undefined. Since the relative projection of an exchange relies on the relative projection of each of the exchange's branches, $G \upharpoonright (p,q)$ is undefined.
- Skip: $C = \mathsf{skip}\ .\ C'$. By the IH, $C'[G_s] \upharpoonright (p,q)$ is undefined. Since the relative projection of a skip relies on the relative projection of the skip's continuation, $G \upharpoonright (p,q)$ is undefined.
- Recursive definition: $C = \mu X\ .\ C'$. It follows from Lemma 12 that $\mathsf{active}(C,G) \subseteq \mathsf{recactive}(X,G)$. Hence, $(p,q) \in \mathsf{recactive}(X,G)$, and thus $(\mu X\ .\ \mathsf{recdef}(X,G)) \upharpoonright (p,q) = (\mu X\ .\ C'[G_s]) \upharpoonright (p,q) \neq \bullet$, which means that $C'[G_s] \upharpoonright (p,q)$ is defined. This contradicts the IH.

The proof for the definedness of local projection is analogous. □

Recall Example 7, where nested recursive definitions in a protocol require nested unfolding of recursive types. The following definition gives us a concise way of writing such nested (or *deep*) unfoldings:

Definition 34 *(Deep unfolding).* Suppose given a sequence of tuples $\tilde{U}$, with each tuple consisting of a recursion variable X_i, a lift $t_i \in \mathbb{N}$, and a type B_i. The *deep unfolding* of the type A with $\tilde{U}$, denoted $\mathsf{deepUnfold}(A,\tilde{U})$, is the type defined as follows:

$$\mathsf{deepUnfold}(A,()) := A$$
$$\mathsf{deepUnfold}(A,(\tilde{U},(X,t,B))) := \mathsf{deepUnfold}(A,\tilde{U})\{\big(\mu X\ .\ (\uparrow^t \mathsf{deepUnfold}(B,\tilde{U}))\big)/X\}$$

When typing a router's recursive call, the types of the router's endpoints are unfoldings of the types in the recursive context. However, because of the deep unfolding in types, this is far from obvious. The following result connects a particular form of deep unfolding with regular unfolding.

Proposition 15. *Suppose given a type A and a sequence of tuples $\tilde{U}$ consisting of a recursion variable, a lift, and a substitution type. Then,*

$$\mathsf{deepUnfold}(A,(\tilde{U},(X,t,A))) = \mathsf{unfold}^t(\mu X\ .\ \mathsf{deepUnfold}(A,\tilde{U})).$$

Proof. By Definition 34:

$$\begin{aligned}\mathsf{deepUnfold}(A,(\tilde{U},(X,t,A))) &= \mathsf{deepUnfold}(A,\tilde{U})\{\big(\mu X\ .\ (\uparrow^t \mathsf{deepUnfold}(A,\tilde{U}))\big)/X\} \\ &= \mathsf{unfold}^t(\mu X\ .\ \mathsf{deepUnfold}(A,\tilde{U}))\end{aligned}$$ □

Armed with these definitions and results, we can finally state our context-based typability result for routers:

Theorem 16. *Suppose given a closed, relative well-formed global type G. Also, suppose given a global type G_s such that $G_s \preceq_C G$, and a $p \in \mathsf{prt}(G)$ for which there is a $q \in \mathsf{prt}(G)$ such that $(p,q) \in \mathsf{active}(C,G)$. Consider:*

- *the participants with whom p interacts in G_s: $\tilde{q} = \{q \in \mathsf{prt}(G) \mid (p,q) \in \mathsf{active}(C,G)\}$,*
- *the absolute priority of G_s: $\mathsf{o}_C = \mathsf{ctxpri}(C)$,*
- *the sequence of bound recursion variables of G_s: $\widetilde{X_C} = \mathsf{ctxbind}(C)$,*
- *for every $X \in \widetilde{X_C}$:*
  - *the body of the recursive definition on X in G: $G_X = \mathsf{recdef}(X,G)$,*
  - *the participants with whom p interacts in G_X: $\tilde{q}_X = \{q \in \mathsf{prt}(G) \mid (p,q) \in \mathsf{recactive}(X,G)\}$,*
  - *the absolute priority of G_X: $\mathsf{o}_X = \mathsf{varpri}(X,G)$,*

- *the sequence of bound recursion variables of G_X excluding X:* $\widetilde{Y_X} = \mathrm{subbind}(\mu X\,.\,G_X, G)$,
- *the type required for μ_p for a recursive call on X:*

$$A_{X,p} = \mathrm{deepUnfold}(\overline{G_X \downarrow^{\circ_X} p}, (Y, t_Y, \overline{G_Y \downarrow^{\circ_Y} p})_{Y \in \widetilde{Y_X}}),$$

- *the type required for p_q for a recursive call on X:*

$$B_{X,q} = \mathrm{deepUnfold}((\!|G_X \upharpoonright (p,q)|\!)^{\circ_X}_{p)q}, (Y, t_Y, (\!|G_Y \upharpoonright (p,q)|\!)^{\circ_Y}_{p)q})_{Y \in \widetilde{Y_X}}),$$

- *the minimum lift for typing a recursive definition on X:* $t_X = \max_{\mathsf{pr}}\left(A_X, (B_{X,q})_{q \in \tilde{q}_X}\right) + 1$,

- *the type expected for μ_p for p's router for G_s:*

$$D_p = \mathrm{deepUnfold}(\overline{G_s \downarrow^{\circ_C} p}, (X, t_X, \overline{G_X \downarrow^{\circ_X} p})_{X \in \widetilde{X_C}}),$$

- *the type expected for p_q for p's router for G_s:*

$$E_q = \mathrm{deepUnfold}((\!|G_s \upharpoonright (p,q)|\!)^{\circ_C}_{p)q}, (X, t_X, (\!|G_X \upharpoonright (p,q)|\!)^{\circ_X}_{p)q})_{X \in \widetilde{X_C}}).$$

Then, we have:

$$[\![G_s]\!]^{\tilde{q}}_p \vdash \left(X : \left(A_X, (B_{X,q})_{q \in \tilde{q}_X}\right)\right)_{X \in \widetilde{X_C}};\ \mu_p : D_p,\ (p_q : E_q)_{q \in \tilde{q}}$$

Proof. We apply induction on the structure of G_s, with six cases as in Algorithm 1. We only detail the cases of exchange and recursion. Axiom Alarm is used in only one sub-case (case 3(c), cf. Fig. 11 below).

- *Exchange*: $G_s = s \twoheadrightarrow r\{i\langle S_i\rangle\,.\,G_i\}_{i\in I}$ (line 3).
  In this case, we add connectives to the types obtained from the IH. Since we do not introduce any recursion variables to these types, the substitutions in the types from the IH are not affected. Hence, we can omit these substitutions from the types. Also, for each $i \in I$, we have $\mathrm{frv}(G_i) \subseteq \mathrm{frv}(G_s)$, i.e. the recursive context remains untouched in this derivation, so we also omit the recursive context.
  Let $\mathsf{deps} := \{q \in \tilde{q} \mid \mathrm{hdep}(q, p, G_s)\}$ (as on line 4). There are three cases depending on the involvement of p.
  1. If $p = s$, then p is the sender (line 5).
     Let us consider the relative projections onto p and the participants in $\tilde{q}$. For the recipient r,

$$G_s \upharpoonright (p,r) = p\{i\,.\,(G_i \upharpoonright (p,r))\}_{i\in I}. \tag{3}$$

     For each $q \in \mathsf{deps}$, by Definition 19, $\mathrm{ddep}((q,p), G) \neq \mathsf{skip}\,.\,R$ for some R. That is, since p is the sender of the exchange, for each $q \in \mathsf{deps}$, by the definitions in Fig. 5,

$$G_s \upharpoonright (p,q) = p!r\{i\,.\,(G_i \upharpoonright (p,q))\}_{i\in I}. \tag{4}$$

     On the other hand, for each $q \in \tilde{q} \setminus \mathsf{deps} \setminus \{r\}$,

$$G_s \upharpoonright (p,q) = \mathsf{skip}\,.\,(G_{i'} \upharpoonright (p,q)) \tag{5}$$

     for any $i' \in I$, because for each $i, j \in I$,

$$G_i \upharpoonright (p,q) = G_j \upharpoonright (p,q). \tag{6}$$

     Let us take stock of the types we expect for each of the router's channels.

     For μ_p we expect
$$\begin{aligned}\overline{G_s \downarrow^{\circ_C} p} &= \overline{\oplus^{\circ_C}\{i : (\!|S_i|\!) \otimes^{\circ_C+1} (G_i \downarrow^{\circ_C+4} p)\}_{i\in I}} \\ &= \&^{\circ_C}\{i : \overline{(\!|S_i|\!)} \parr^{\circ_C+1} \overline{(G_i \downarrow^{\circ_C+4} p)}\}_{i\in I}. \end{aligned} \tag{7}$$

     For p_r we expect
$$\begin{aligned}(\!|G_s \upharpoonright (p,q)|\!)^{\circ_C}_{p)r} &= (\!|p\{i\,.\,(G_i \upharpoonright (p,r))\}_{i\in I}|\!)^{\circ_C}_{p)r} && \text{(cf. (3))} \\ &= \oplus^{\circ_C+1}\{i : (\!|S_i|\!) \otimes^{\circ_C+2} (\!|G_i \upharpoonright (p,r)|\!)^{\circ_C+4}_{p)r}\}_{i\in I}. && (8)\end{aligned}$$

     For each $q \in \mathsf{deps}$,

     for p_q we expect
$$\begin{aligned}(\!|G_s \upharpoonright (p,q)|\!)^{\circ_C}_{p)q} &= (\!|p!r\{i.\,(G_i \upharpoonright (p,q))\}_{i\in I}|\!)^{\circ_C}_{p)q} && \text{(cf. (4))} \\ &= \oplus^{\circ_C+1}\{i : (\!|G_i \upharpoonright (p,q)|\!)^{\circ_C+4}_{p)q}\}_{i\in I}. && (9)\end{aligned}$$

     For each $q \in \tilde{q} \setminus \mathsf{deps} \setminus \{r\}$,

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall i \in I.\ v \leftrightarrow w \vdash v : \overline{(\!|S_i|\!)}, w : (\!|S_i|\!)}\ \textsc{Id} \quad \forall i \in I.\ [\![G_i]\!]_p^{\tilde{q}} \vdash \mu_p : \overline{(G_i \downarrow^{\mathsf{o_C}+4} p)}, \left(p_q : (\!|G_i \upharpoonright (p,q)|\!)_{p\rangle q}^{\mathsf{o_C}+4}\right)_{q\in\tilde{q}}}{\forall i \in I.\ v \leftrightarrow w \mid [\![G_i]\!]_p^{\tilde{q}} \vdash \mu_p : \overline{(G_i \downarrow^{\mathsf{o_C}+4} p)}, v : \overline{(\!|S_i|\!)}, w : (\!|S_i|\!), \left(p_q : (\!|G_i \upharpoonright (p,q)|\!)_{p\rangle q}^{\mathsf{o_C}+4}\right)_{q\in\tilde{q}}}\ \textsc{Mix}}{\forall i \in I.\ 5_i \vdash \mu_p : \overline{(G_i \downarrow^{\mathsf{o_C}+4} p)}, v : \overline{(\!|S_i|\!)}, p_r : (\!|S_i|\!) \otimes^{\mathsf{o_C}+2} (\!|G_i \upharpoonright (p,r)|\!)_{p\rangle r}^{\mathsf{o_C}+4}, \left(p_q : (\!|G_i \upharpoonright (p,q)|\!)_{p\rangle q}^{\mathsf{o_C}+4}\right)_{q\in\tilde{q}\setminus\{r\}}}\ \otimes^\star}{\forall i \in I.\ 4_i \vdash \mu_p : \overline{(\!|S_i|\!)} \parr^{\mathsf{o_C}+1} \overline{(G_i \downarrow^{\mathsf{o_C}+4} p)}, p_r : (\!|S_i|\!) \otimes^{\mathsf{o_C}+2} (\!|G_i \upharpoonright (p,r)|\!)_{p\rangle r}^{\mathsf{o_C}+4}, \left(p_q : (\!|G_i \upharpoonright (p,q)|\!)_{p\rangle q}^{\mathsf{o_C}+4}\right)_{q\in\tilde{q}\setminus\{r\}}}\ \parr}{\forall i \in I.\ 3_i \vdash \mu_p : \overline{(\!|S_i|\!)} \parr^{\mathsf{o_C}+1} \overline{(G_i \downarrow^{\mathsf{o_C}+4} p)}, p_r : (\!|S_i|\!) \otimes^{\mathsf{o_C}+2} (\!|G_i \upharpoonright (p,r)|\!)_{p\rangle r}^{\mathsf{o_C}+4}, \left(p_q : \oplus^{\mathsf{o_C}+1}\{i : (\!|G_i \upharpoonright (p,q)|\!)_{p\rangle q}^{\mathsf{o_C}+4}\}_{i\in I}\right)_{q\in\mathsf{deps}}, \left(p_q : (\!|G_i \upharpoonright (p,q)|\!)_{p\rangle q}^{\mathsf{o_C}+4}\right)_{q\in\tilde{q}\setminus\mathsf{deps}}}\ (\oplus^\star)^*}{\forall i \in I.\ 2_i \vdash \mu_p : \overline{(\!|S_i|\!)} \parr^{\mathsf{o_C}+1} \overline{(G_i \downarrow^{\mathsf{o_C}+4} p)}, p_r : \oplus^{\mathsf{o_C}+1}\{i : (\!|S_i|\!) \otimes^{\mathsf{o_C}+2} (\!|G_i \upharpoonright (p,r)|\!)_{p\rangle r}^{\mathsf{o_C}+4}\}_{i\in I}, \left(p_q : \oplus^{\mathsf{o_C}+1}\{i : (\!|G_i \upharpoonright (p,q)|\!)_{p\rangle q}^{\mathsf{o_C}+4}\}_{i\in I}\right)_{q\in\mathsf{deps}}, \left(p_q : (\!|G_i \upharpoonright (p,q)|\!)_{p\rangle q}^{\mathsf{o_C}+4}\right)_{q\in\tilde{q}\setminus\mathsf{deps}}}\ \oplus^\star \quad (\text{cf. (6)})}{\begin{aligned}[\![G_s]\!]_p^{\tilde{q}} = 1 \vdash \mu_p &: \&^{\mathsf{o_C}}\{i : \overline{(\!|S_i|\!)} \parr^{\mathsf{o_C}+1} \overline{(G_i \downarrow^{\mathsf{o_C}+4} p)}\}_{i\in I}, && (\text{cf. (7)})\\ p_r &: \oplus^{\mathsf{o_C}+1}\{i : (\!|S_i|\!) \otimes^{\mathsf{o_C}+2} (\!|G_i \upharpoonright (p,r)|\!)_{p\rangle r}^{\mathsf{o_C}+4}\}_{i\in I}, && (\text{cf. (8)})\\ \left(p_q : \oplus^{\mathsf{o_C}+1}\{i : (\!|G_i \upharpoonright (p,q)|\!)_{p\rangle q}^{\mathsf{o_C}+4}\}_{i\in I}\right)_{q\in\mathsf{deps}}, && (\text{cf. (9)})\\ \left(p_q : (\!|G_{i'} \upharpoonright (p,q)|\!)_{p\rangle q}^{\mathsf{o_C}+4}\right)_{q\in\tilde{q}\setminus\mathsf{deps}} && (\text{cf. (10)})\end{aligned}}\ \parr$$

Fig. 10. Typing derivation used in the proof of Theorem 11.

for p_q we expect
$$(\!|G_s \upharpoonright (p,q)|\!)_{p\rangle q}^{\mathsf{o_C}} = (\!|\mathsf{skip} \,.\, (G_{i'} \upharpoonright (p,q))|\!)_{p\rangle q}^{\mathsf{o_C}} \quad (\text{cf. (5)})$$
$$= (\!|G_{i'} \upharpoonright (p,q)|\!)_{p\rangle q}^{\mathsf{o_C}+4} \text{ for any } i' \in I. \quad (10)$$

Let us now consider the process returned by Algorithm 1, with each prefix marked with a number:

$$[\![G_s]\!]_p^{\tilde{q}} = \underbrace{\mu_p \triangleright \{i :}_{1}\ \underbrace{\overline{p_r} \triangleleft i}_{2_i} \cdot \underbrace{(\overline{p_q} \triangleleft i)_{q\in\mathsf{deps}}}_{3_i} \cdot \underbrace{\mu_p(v)}_{4_i} \,.\, \underbrace{\overline{p_r}[w]}_{5_i} \cdot (v \leftrightarrow w \mid [\![G_i]\!]_p^{\tilde{q}})\}_{i\in I}$$

For each $i' \in I$, let $C_{i'} := C[s \twoheadrightarrow r(\{i\langle S_i\rangle \,.\, G_i\}_{i\in I\setminus\{i'\}} \cup \{i'\langle S_{i'}\rangle \,.\, []\})]$. Clearly, $G_{i'} \preceq_{C_{i'}} G$. Also, because we are not adding recursion binders, the current value of $\tilde{q}$ is appropriate for the IH. With this context $C_{i'}$ and $\tilde{q}$, we apply the IH to obtain the typing of $[\![G_{i'}]\!]_p^{\tilde{q}}$, where priorities start at $\mathrm{ctxpri}(C_{i'}) = \mathrm{ctxpri}(C) + 4 = \mathsf{o_C} + 4$ (cf. Definition 32). Following these typings, Fig. 10 gives the typing of $[\![G_s]\!]_p^{\tilde{q}}$, referring to parts of the process by the number marking its foremost prefix above.

Clearly, the priorities in the derivation of Fig. 10 meet all requirements. The order of the applications of $\oplus^\star$ for each $q \in \mathsf{deps}$ does not matter, since the selection actions are asynchronous.

2. If $p = r$, then p is the recipient (line 6). This case is analogous to the previous one.
3. If $p \notin \{r, s\}$ (line 7), then further analysis depends on whether the exchange is a dependency for p. Let

$$\mathsf{depon}_s := (s \in \tilde{q} \wedge \mathrm{hdep}(p, s, G)) \quad \text{(as on line 8), and}$$
$$\mathsf{depon}_r := (r \in \tilde{q} \wedge \mathrm{hdep}(p, r, G)) \quad \text{(as on line 9).}$$

To see what the truths of depon_s and depon_r mean, we follow Definition 19 and the definitions in Fig. 5.

$$G_s \upharpoonright (p,s) = \begin{cases} s!r\{i \,.\, (G_i \upharpoonright (p,s))\}_{i\in I} & \text{if } \mathsf{depon}_s \text{ is true} \\ \mathsf{skip} \,.\, (G_{i'} \upharpoonright (p,s)) \text{ for any } i' \in I & \text{otherwise} \end{cases} \quad (11)$$

$$G_s \upharpoonright (p,r) = \begin{cases} r?s\{i\,.\,(G_i \upharpoonright (p,r))\}_{i\in I} & \text{if } \mathsf{depon}_r \text{ is true} \\ \mathsf{skip}\,.\,(G_{i'} \upharpoonright (p,r)) \text{ for any } i' \in I & \text{otherwise} \end{cases} \tag{12}$$

Let us also consider the relative projections onto p and the participants in $\tilde{q}$ besides r and s, which follow by the relative well-formedness of G_s. For each $q \in \tilde{q} \setminus \{r,s\}$,

$$G_s \upharpoonright (p,q) = \mathsf{skip}\,.\,(G_{i'} \upharpoonright (p,q)) \tag{13}$$

for any $i' \in I$.

The rest of the analysis depends on the truth of depon_s and depon_r. There are four cases.

(a) If depon_s is true and depon_r is false (line 10), let us take stock of the types we expect for each of the router's channels.

For μ_p we expect
$$\overline{G_s \downarrow^{o_C} p} = \overline{\&^{o_C+2}\{i : (G_i \downarrow^{o_C+4} p)\}_{i\in I}} = \oplus^{o_C+2}\{i : \overline{(G_i \downarrow^{o_C+4} p)}\}_{i\in I}. \tag{14}$$

For p_s we expect
$$(\!|G_s \upharpoonright (p,s)|\!)^{o_C}_{p\rangle s} = (\!|s!r\{i\,.\,(G_i \upharpoonright (p,s))\}_{i\in I}|\!)^{o_C}_{p\rangle s} \quad \text{(cf. (11))}$$
$$= \&^{o_C+1}\{i : (\!|G_i \upharpoonright (p,s)|\!)^{o_C+4}_{p\rangle s}\}_{i\in I}. \tag{15}$$

For each $q \in \tilde{q} \setminus \{s\}$,

for p_q we expect
$$(\!|G_s \upharpoonright (p,q)|\!)^{o_C}_{p\rangle q} = (\!|\mathsf{skip}\,.\,(G_{i'} \upharpoonright (p,q))|\!)^{o_C}_{p\rangle q} \quad \text{(cf. (12) and (13))}$$
$$= (\!|G_{i'} \upharpoonright (p,q)|\!)^{o_C+4}_{p\rangle q} \text{ for any } i' \in I. \tag{16}$$

Similar to case (1), we apply the IH to obtain the typing of $[\![G_i]\!]^{\tilde{q}}_p$ for each $i \in I$, starting at priority $o_C + 4$. We derive the typing of $[\![G_s]\!]^{\tilde{q}}_p$:

$$\dfrac{\dfrac{\forall i \in I.\ [\![G_i]\!]^{\tilde{q}}_p \vdash \mu_p : \overline{G_i \downarrow^{o_C+4} p},\ \left(p_q : (\!|G_i \upharpoonright (p,q)|\!)^{o_C+4}_{p\rangle q}\right)_{q\in\tilde{q}}}{\forall i \in I.\ \overline{\mu_p} \triangleleft i \cdot [\![G_i]\!]^{\tilde{q}}_p \vdash \mu_p : \oplus^{o_C+2}\{i : \overline{G_i \downarrow^{o_C+4} p}\}_{i\in I},\ \left(p_q : (\!|G_i \upharpoonright (p,q)|\!)^{o_C+4}_{p\rangle q}\right)_{q\in\tilde{q}}}\ \oplus^\star}{[\![G_s]\!]^{\tilde{q}}_p = p_s \triangleright \{i : \overline{\mu_p} \triangleleft i \cdot [\![G_i]\!]^{\tilde{q}}_p\}_{i\in I} \vdash \mu_p : \oplus^{o_C+2}\{i : \overline{G_i \downarrow^{o_C+4} p}\}_{i\in I},\ \text{(cf. (14))}\quad p_s : \&^{o_C+1}\{i : (\!|G_i \upharpoonright (p,q)|\!)^{o_C+4}_{p\rangle q}\}_{i\in I},\ \text{(cf. (15))}\quad \left(p_q : (\!|G'_i \upharpoonright (p,q)|\!)^{o_C+4}_{p\rangle q}\right)_{q\in\tilde{q}}\ \text{(cf. (16))}}\ \otimes$$

(b) The case where depon_s is false and depon_r is true (line 11) is analogous to the previous one.

(c) If both depon_s and depon_r are true (line 12 and (2)), let us once again take stock of the types we expect for each of the router's channels.

For μ_p we expect
$$\overline{G_s \downarrow^{o_C} p} = \overline{\&^{o_C+2}\{i : (G_i \downarrow^{o_C+4} p)\}_{i\in I}} = \oplus^{o_C+2}\{i : \overline{(G_i \downarrow^{o_C+4} p)}\}_{i\in I} \tag{17}$$

For p_s we expect
$$(\!|G_s \upharpoonright (p,s)|\!)^{o_C}_{p\rangle s} = (\!|s!r\{i\,.\,(G_i \upharpoonright (p,s))\}_{i\in I}|\!)^{o_C}_{p\rangle s} \quad \text{(cf. (11))}$$
$$= \&^{o_C+1}\{i : (\!|G_i \upharpoonright (p,s)|\!)^{o_C+4}_{p\rangle s}\}_{i\in I} \tag{18}$$

For p_r we expect
$$(\!|G_s \upharpoonright (p,r)|\!)^{o_C}_{p\rangle r} = (\!|r?s\{i\,.\,(G_i \upharpoonright (p,r))\}_{i\in I}|\!)^{o_C}_{p\rangle r} \quad \text{(cf. (12))}$$
$$= \&^{o_C+2}\{i : (\!|G_i \upharpoonright (p,r)|\!)^{o_C+4}_{p\rangle r}\}_{i\in I} \tag{19}$$

For each $q \in \tilde{q} \setminus \{s,r\}$,

for p_q we expect
$$(\!|G_s \upharpoonright (p,q)|\!)^{o_C}_{p\rangle q} = (\!|\mathsf{skip}\,.\,(G_{i'} \upharpoonright (p,q))|\!)^{o_C}_{p\rangle q} \quad \text{(cf. (13))}$$
$$= (\!|G_{i'} \upharpoonright (p,q)|\!)^{o_C+4}_{p\rangle q} \text{ for any } i' \in I \tag{20}$$

It is clear from (18) and (19) that the router will receive label $i \in I$ first on p_s and then $i' \in I$ on p_r. We rely on alarm processes (Definition 28) to handle the case $i' \neq i$.

Similar to case (1), we apply the IH to obtain the typing of $[\![G_i]\!]^{\tilde{q}}_p$ for each $i \in I$, starting at priority $o_C + 4$. Fig. 11 gives the typing of $[\![G_s]\!]^{\tilde{q}}_p$.

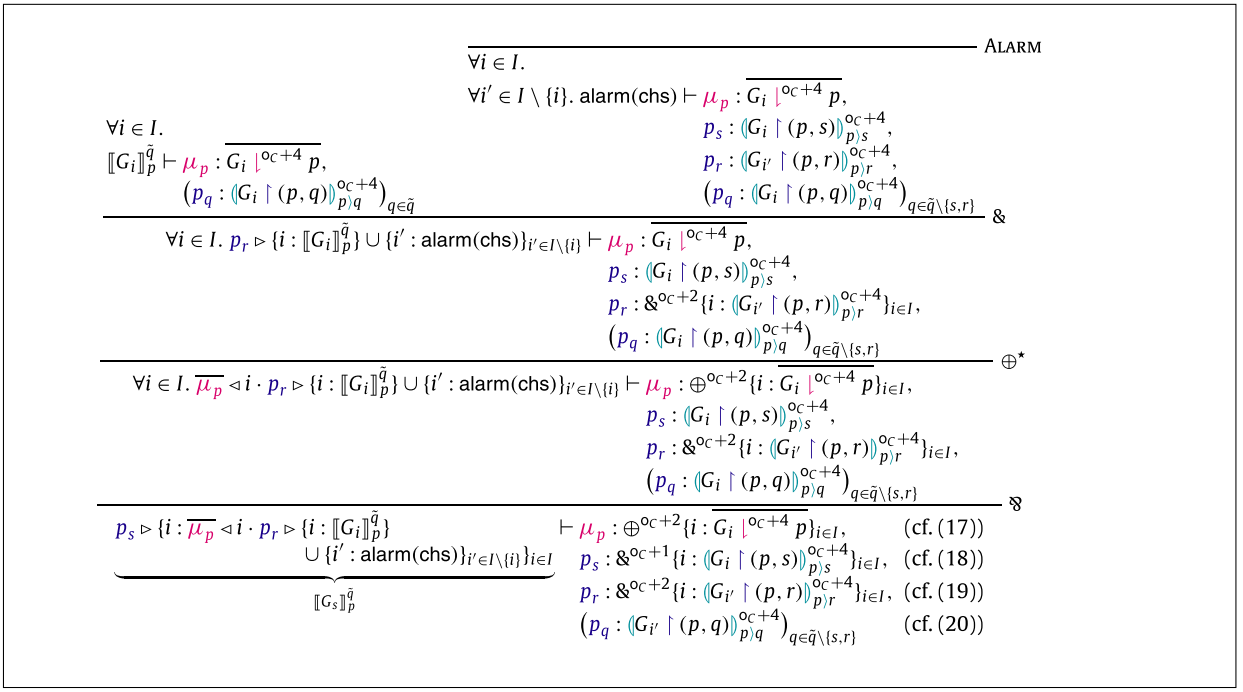

Fig. 11. Typing derivation used in the proof of Theorem 11, where $\mathsf{chs} = \{\mu_p\} \cup \{p_q \mid q \in \tilde{q}\}$.

(d) If both depon_s and depon_r are false, let us again take stock of the types we expect for each of the router's channels.

For μ_p we expect $\quad \overline{G_s \lfloor^{\mathsf{o}_C} p} = \overline{G_{i'} \lfloor^{\mathsf{o}_C+4} p}$ for any $i' \in I$.

For each $q \in \tilde{q}$,

for p_q we expect $\quad (\!|G_s \upharpoonright (p,q)|\!)^{\mathsf{o}_C}_{p\rangle q} = (\!|\mathsf{skip} \,.\, (G_{i'} \upharpoonright (p,q))|\!)^{\mathsf{o}_C}_{p\rangle q}$ (cf. (11), (12) and (13))
$= (\!|G_{i'} \upharpoonright (p,q)|\!)^{\mathsf{o}_C+4}_{p\rangle q}$ for any $i' \in I$.

Similar to case (1), we apply the IH to obtain the typing of $[\![G_{i'}]\!]^{\tilde{q}}_p$, starting at priority $\mathsf{o}_C + 4$. This directly proves the thesis.

- *Recursive definition*: $G_s = \mu Z \,.\, G'$ (line 14).
Let

$$\tilde{q}' := \{q \in \tilde{q} \mid G_s \upharpoonright (p,q) \neq \bullet\} \tag{21}$$

(as on line 15). We consider the relative projections onto p and the participants in $\tilde{q}$. For each $q \in \tilde{q}'$, we know $G_s \upharpoonright (p,q) \neq \bullet$, while for each $q \in \tilde{q} \setminus \tilde{q}'$, we know $G_s \upharpoonright (p,q) = \bullet$. More precisely, by Definition 17, for each $q \in \tilde{q}'$,

$$G_s \upharpoonright (p,q) = (\mu Z \,.\, G') \upharpoonright (p,q) = \mu Z \,.\, (G' \upharpoonright (p,q)), \tag{22}$$

and thus

$$G' \upharpoonright (p,q) \neq \mathsf{skip}^* \,.\, \bullet \text{ and } G' \upharpoonright (p,q) \neq \mathsf{skip}^* \,.\, Z.$$

For each $q \in \tilde{q} \setminus \tilde{q}'$,

$$G_s \upharpoonright (p,q) = (\mu Z \,.\, G') \upharpoonright (p,q) = \bullet, \tag{23}$$

and thus

$$G' \upharpoonright (p,q) = \mathsf{skip}^* \,.\, \bullet \text{ or } G' \upharpoonright (p,q) = \mathsf{skip}^* \,.\, Z.$$

Further analysis depends on whether $\tilde{q}' = \emptyset$ or not. We thus examine two cases:

– If $\tilde{q}' = \emptyset$ (line 16), let us consider the local projection $G_s \lfloor^{\circ_C} p$. We prove that $G_s \lfloor^{\circ_C} p = \bullet$.
Suppose, for contradiction, that $G_s \lfloor^{\circ_C} p \neq \bullet$. Then, by the definitions in Fig. 7, $G' \lfloor^{\circ_C} p \neq X$ and $G' \lfloor^{\circ_C} p \neq \bullet$. That is, $G' \lfloor^{\circ_C} p$ contains communication actions or some recursion variable other than Z. However, communication actions in $G' \lfloor^{\circ_C} p$ originate from exchanges in G', either involving p and some $q \in \tilde{q}$, or as a dependency on an exchange involving some $q \in \tilde{q}$. Moreover, recursion variables in $G' \lfloor^{\circ_C} p$ originate from recursion variables in G'. But this would mean that for this q, $G' \upharpoonright (p, q)$ contains interactions or recursion variables, contradicting (23). Therefore, it cannot be the case that $G_s \lfloor^{\circ_C} p \neq \bullet$.
Let us take stock of the types we expect for each of the router's channels. For now, we omit the substitutions in the types.

$$\text{For } \mu_p \text{ we expect} \qquad \overline{G_s \lfloor^{\circ_C} p} = \overline{\bullet} = \bullet.$$

$$\text{For each } q \in \tilde{q}, \text{ for } p_q \text{ we expect} \qquad (\!|G_s \upharpoonright (p,q)|\!)^{\circ_C}_{p \rangle q} = (\!|\bullet|\!)^{\circ_C}_{p \rangle q} = \bullet. \qquad \text{(cf. (23))}$$

Because all expected types are $\bullet$, the substitutions do not affect the types, so we can omit them altogether.
First we apply Empty, giving us an arbitrary recursive context, and thus the recursive context we need. Then, we apply $\bullet$ for μ_p and for p_q for each $q \in \tilde{q}$, and obtain the typing of $[\![G_s]\!]^{\tilde{q}}_p$ (omitting the recursive context):

$$[\![G_s]\!]^{\tilde{q}}_p = \mathbf{0} \vdash \mu_p : \bullet, (p_q : \bullet)_{q \in \tilde{q}}$$

– If $\tilde{q}' \neq \emptyset$ (line 17), then, following similar reasoning as in the previous case, $G_s \lfloor^{\circ_C} p = \mu Z \,.\, (G' \lfloor^{\circ_C} p)$. We take stock of the types we expect for each of the router's channels. Note that, because of the recursive definition on Z in G_s, there cannot be another recursive definition in the context C capturing the recursion variable Z. Therefore, by Definition 30, $Z \notin \widetilde{X_C}$.

$$\begin{aligned}
&\text{For } \mu_p \text{ we expect} && \text{deepUnfold}(\overline{G_s \lfloor^{\circ_C} p}, \ldots) \\
&&& = \text{deepUnfold}(\overline{\mu Z \,.\, (G' \lfloor^{\circ_C} p)}, \ldots) \\
&&& = \text{deepUnfold}(\mu Z \,.\, \overline{G' \lfloor^{\circ_C} p}, \ldots) \\
&&& = \mu Z \,.\, \text{deepUnfold}(\overline{G' \lfloor^{\circ_C} p}, (X, t_X, \overline{G_X \lfloor^{\circ_X} p})_{X \in \widetilde{X_C}}). && (24) \\
&\text{For each } q \in \tilde{q}', && \\
&\text{for } p_q \text{ we expect} && \text{deepUnfold}((\!|G_s \upharpoonright (p,q)|\!)^{\circ_C}_{p \rangle q}, \ldots) \\
&&& = \text{deepUnfold}((\!|\mu Z \,.\, (G' \upharpoonright (p,q))|\!)^{\circ_C}_{p \rangle q}, \ldots) \\
&&& = \text{deepUnfold}(\mu Z \,.\, (\!|G' \upharpoonright (p,q)|\!)^{\circ_C}_{p \rangle q}, \ldots) && \text{(cf. (22))} \\
&&& = \mu Z \,.\, \text{deepUnfold}((\!|G' \upharpoonright (p,q)|\!)^{\circ_C}_{p \rangle q}, (X, t_X, (\!|G_X \upharpoonright (p,q)|\!)^{\circ_X}_{p \rangle q})_{X \in \widetilde{X_C}}). && (25) \\
&\text{For each } q \in \tilde{q} \setminus \tilde{q}', && \\
&\text{for } p_q \text{ we expect} && \text{deepUnfold}((\!|G_s \upharpoonright (p,q)|\!)^{\circ_C}_{p \rangle q}, \ldots) \\
&&& = \text{deepUnfold}((\!|\bullet|\!)^{\circ_C}_{p \rangle q}, \ldots) \\
&&& = \text{deepUnfold}(\bullet, \ldots) && \text{(cf. (23))} \\
&&& = \bullet. && (26)
\end{aligned}$$

We also need an assignment in the recursive context for every $X \in \widetilde{X_C}$, but not for Z.
Let $C' = C[\mu Z \,.\, []]$. Clearly, $G' \leq_{C'} G$. Let us first establish some facts about the recursion binders, priorities, and active participants related to C', G', and Z:
* $\widetilde{X_{C'}} = \text{ctxbind}(C') = (\text{ctxbind}(C), Z) = (\widetilde{X_C}, Z)$ (cf. Definition 30).
* $G_Z = \text{recdef}(Z, G) = G'$, as proven by the context C' (cf. Definition 31).
* $\widetilde{Y_Z} = \text{subbind}(\mu Z \,.\, G_Z, G) = \text{ctxbind}(C) = \widetilde{X_C}$.
* $\circ_{C'} = \text{ctxpri}(C') = \text{ctxpri}(C) = \circ_C$, and $\circ_Z = \text{varpri}(Z, G) = \text{ctxpri}(C) = \circ_C$, and hence $\circ_{C'} = \circ_Z$ (cf. Definition 32).
* $\tilde{q}_Z = \tilde{q}'$ (cf. Definition 33 and (21)).
Because $\widetilde{X_{C'}} = (\widetilde{X_C}, Z)$ and $\tilde{q}' = \tilde{q}_Z$, $\tilde{q}'$ is appropriate for the IH. We apply the IH on C', G', and $\tilde{q}'$ to obtain a typing for $[\![G']\!]^{\tilde{q}'}_p$, where we immediately make use of the facts established above. We give the assignment to Z in the recursive context separate from those for the recursion variables in $\widetilde{X_C}$. Also, by Proposition 15, we can write the final unfolding on Z in the types separately. For example, the type for μ_p is

$$
\begin{aligned}
&\mathrm{deepUnfold}(\overline{G' \lfloor^{o_{C'}} p}, (X, t_X, \overline{G_X \lfloor^{o_X} p})_{X \in \widetilde{X_{C'}}}) \\
&= \mathrm{deepUnfold}(\overline{G' \lfloor^{o_C} p}, (X, t_X, \overline{G_X \lfloor^{o_X} p})_{X \in (\widetilde{X_C}, Z)}) \\
&= \mathrm{deepUnfold}(\overline{G' \lfloor^{o_C} p}, \big((X, t_X, \overline{G_X \lfloor^{o_X} p})_{X \in \widetilde{X_C}}, (Z, t_Z, \overline{G_Z \lfloor^{o_Z} p})\big)) \\
&= \mathrm{deepUnfold}(\overline{G' \lfloor^{o_C} p}, \big((X, t_X, \overline{G_X \lfloor^{o_X} p})_{X \in \widetilde{X_C}}, (Z, t_Z, \overline{G' \lfloor^{o_C} p})\big)) \\
&= \mathrm{unfold}^{t_Z}\big(\mu Z \,.\, \mathrm{deepUnfold}(\overline{G' \lfloor^{o_C} p}, (X, t_X, \overline{G_X \lfloor^{o_X} p})_{X \in \widetilde{X_C}})\big).
\end{aligned}
$$

The resulting typing is as follows:

$$
\begin{aligned}
[\![G']\!]^{\tilde{q}'}_p \vdash &\left(X : \left(\begin{array}{l} \mathrm{deepUnfold}(\overline{G_X \lfloor^{o_X} p}, (Y, t_Y, \overline{G_Y \lfloor^{o_Y} p})_{Y \in \widetilde{Y_X}}), \\ \left(\mathrm{deepUnfold}(\langle\!| G_X \restriction (p,q) |\!\rangle^{o_X}_{p \rangle q}, (Y, t_Y, \langle\!| G_Y \restriction (p,q) |\!\rangle^{o_Y}_{p \rangle q})_{Y \in \widetilde{Y_X}}) \right)_{q \in \tilde{q}_X} \end{array} \right) \right)_{X \in \widetilde{X_C}}, \\
&Z : \left(\begin{array}{l} \mathrm{deepUnfold}(\overline{G' \lfloor^{o_C} p}, (X, t_X, \overline{G_X \lfloor^{o_X} p})_{X \in \widetilde{X_C}}), \\ \left(\mathrm{deepUnfold}(\langle\!| G' \restriction (p,q) |\!\rangle^{o_C}_{p \rangle q}, (X, t_X, \langle\!| G_X \restriction (p,q) |\!\rangle^{o_X}_{p \rangle q})_{X \in \widetilde{X_C}}) \right)_{q \in \tilde{q}'} \end{array} \right); \\
&\mu_p : \mathrm{unfold}^{t_Z}\big(\mu Z \,.\, \mathrm{deepUnfold}(\overline{G' \lfloor^{o_C} p}, (X, t_X, \overline{G_X \lfloor^{o_X} p})_{X \in \widetilde{X_C}})\big), \\
&\left(p_q : \mathrm{unfold}^{t_Z}\big(\mu Z \,.\, \mathrm{deepUnfold}(\langle\!| G' \restriction (p,q) |\!\rangle^{o_C}_{p \rangle q}, (X, t_X, \langle\!| G_X \restriction (p,q) |\!\rangle^{o_X}_{p \rangle q})_{X \in \widetilde{X_C}})\big) \right)_{q \in \tilde{q}'}
\end{aligned}
$$

By assumption, we have

$$
t_Z = \max_{\mathrm{pr}} \left(\begin{array}{l} \mathrm{deepUnfold}(\overline{G' \lfloor^{o_C} p}, (X, t_X, \overline{G_X \lfloor^{o_X} p})_{X \in \widetilde{X_C}}) \\ \left(\mathrm{deepUnfold}(\langle\!| G' \restriction (p,q) |\!\rangle^{o_C}_{p \rangle q}, (X, t_X, \langle\!| G_X \restriction (p,q) |\!\rangle^{o_X}_{p \rangle q})_{X \in \widetilde{X_C}}) \right)_{q \in \tilde{q}'} \end{array} \right) + 1,
$$

so t_Z is clearly greater than the maximum priority appearing in the types before unfolding. Hence, we can apply Rec to eliminate Z from the recursive context, and to fold the types, giving the typing of $[\![G_s]\!]^{\tilde{q}}_p = \mu Z(\mu_p, (p_q)_{q \in \tilde{q}'}) \,.\, [\![G']\!]^{\tilde{q}'}_p$:

$$
\begin{aligned}
[\![G_s]\!]^{\tilde{q}}_p \vdash &\left(X : \left(\begin{array}{l} \mathrm{deepUnfold}(\overline{G_X \lfloor^{o_X} p}, (Y, t_Y, \overline{G_Y \lfloor^{o_Y} p})_{Y \in \widetilde{Y_X}}), \\ \left(\mathrm{deepUnfold}(\langle\!| G_X \restriction (p,q) |\!\rangle^{o_X}_{p \rangle q}, (Y, t_Y, \langle\!| G_Y \restriction (p,q) |\!\rangle^{o_Y}_{p \rangle q})_{Y \in \widetilde{Y_X}}) \right)_{q \in \tilde{q}_X} \end{array} \right) \right)_{X \in \widetilde{X_C}}; \\
&\mu_p : \mu Z \,.\, \mathrm{deepUnfold}(\overline{G' \lfloor^{o_C} p}, (X, t_X, \overline{G_X \lfloor^{o_X} p})_{X \in \widetilde{X_C}}), \\
&\left(p_q : \mu Z \,.\, \mathrm{deepUnfold}(\langle\!| G' \restriction (p,q) |\!\rangle^{o_C}_{p \rangle q}, (X, t_X, \langle\!| G_X \restriction (p,q) |\!\rangle^{o_X}_{p \rangle q})_{X \in \widetilde{X_C}}) \right)_{q \in \tilde{q}'}
\end{aligned}
$$

In this typing, the type for μ_p concurs with (24), and, for every $q \in \tilde{q}'$, the type for p_q concurs with (25). For every $q \in \tilde{q} \setminus \tilde{q}'$, we can add the type for p_q in (26) by applying $\bullet$. This proves the thesis.

- *Recursive call*: $G_s = Z$ (line 18).
  Clearly, because G is closed (i.e. $\mathrm{frv}(G) = \emptyset$), $Z \in \widetilde{X_C}$. More precisely, $\widetilde{X_C} = (\tilde{X}_1, Z, \tilde{X}_2)$.
  Note that the recursive definitions on the variables in $\tilde{X}_1$ appear in G after the recursive definitions on the variables in $(Z, \tilde{X}_2)$. Because the unfoldings of $(Z, \tilde{X}_2)$ occur before the unfoldings of $\tilde{X}_1$, the recursive definitions on the variables in $\tilde{X}_1$ are renamed in order to avoid capturing these variables when performing the unfoldings of $(Z, \tilde{X}_2)$. So, after the unfoldings of $(Z, \tilde{X}_2)$, there are no recursive calls on the variables in $\tilde{X}_1$ anymore, so the unfoldings on $\tilde{X}_1$ do not have any effect on the types.
  Also, note that $\tilde{X}_2 = \widetilde{Y_Z}$ (cf. Definition 30).
  Let us take stock of the types we expect for our router's channels.

$$
\begin{aligned}
\text{For } \mu_p \text{ we expect} \quad & \mathrm{deepUnfold}(\overline{G_s \lfloor^{o_C} p}, \ldots) \\
&= \mathrm{deepUnfold}(\overline{Z \lfloor^{o_C} p}, \ldots) \\
&= \mathrm{deepUnfold}(\overline{Z}, \ldots) \\
&= \mathrm{deepUnfold}(Z, (X, t_X, \overline{G_X \lfloor^{o_X} p})_{X \in (\tilde{X}_1, Z, \widetilde{Y_Z})}) \\
&= \mathrm{deepUnfold}(Z, (X, t_X, \overline{G_X \lfloor^{o_X} p})_{X \in (Z, \widetilde{Y_Z})}) \\
&= \mu Z \,.\, (\uparrow^{t_Z} \mathrm{deepUnfold}(\overline{G_Z \lfloor^{o_Z} p}, (X, t_X, \overline{G_X \lfloor^{o_X} p})_{X \in \widetilde{Y_Z}})) \qquad (27)
\end{aligned}
$$

For each $q \in \tilde{q}$,

$$
\begin{aligned}
\text{for } p_q \text{ we expect} \quad & \mathsf{deepUnfold}(\llparenthesis G_s \upharpoonright (p,q) \rrparenthesis^{\mathsf{o}_C}_{p \rangle q}, \ldots) \\
&= \mathsf{deepUnfold}(\llparenthesis Z \rrparenthesis^{\mathsf{o}_C}_{p \rangle q}, \ldots) \\
&= \mathsf{deepUnfold}(Z, (X, t_X, \llparenthesis G_X \upharpoonright (p,q) \rrparenthesis^{\mathsf{o}_X}_{p \rangle q})_{X \in (\tilde{X}_1, Z, \widetilde{Y_Z})}) \\
&= \mathsf{deepUnfold}(Z, (X, t_X, \llparenthesis G_X \upharpoonright (p,q) \rrparenthesis^{\mathsf{o}_X}_{p \rangle q})_{X \in (Z, \widetilde{Y_Z})}) \\
&= \mu Z \,.\, (\uparrow^{t_Z} \mathsf{deepUnfold}(\llparenthesis G_Z \upharpoonright (p,q) \rrparenthesis^{\mathsf{o}_Z}_{p \rangle q}, (X, t_X, \llparenthesis G_X \upharpoonright (p,q) \rrparenthesis^{\mathsf{o}_X}_{p \rangle q})_{X \in \widetilde{Y_Z}})) \qquad (28)
\end{aligned}
$$

Also, we need an assignment in the recursive context for every $X \in \widetilde{X_C}$. By Lemma 13, $\tilde{q} = \tilde{q}_Z$. Hence, for Z, the assignment should be as follows:

$$
Z : \left(\begin{array}{l} \mathsf{deepUnfold}(\overline{G_Z \lfloor^{\mathsf{o}_Z} p}, (X, t_X, \overline{G_X \lfloor^{\mathsf{o}_X} p})_{X \in \widetilde{Y_Z}}), \\ \left(\mathsf{deepUnfold}(\llparenthesis G_Z \upharpoonright (p,q) \rrparenthesis^{\mathsf{o}_Z}_{p \rangle q}, (X, t_X, \llparenthesis G_X \upharpoonright (p,q) \rrparenthesis^{\mathsf{o}_X}_{p \rangle q})_{X \in \widetilde{Y_Z}} \right)_{q \in \tilde{q}} \end{array} \right) \qquad (29)
$$

We apply VAR to obtain the typing of $[\![G_s]\!]^p_{\tilde{q}}$, where we make us the rule's allowance for an arbitrary recursive context up to the assignment to Z. VAR is applicable, because the types are recursive definitions on Z, concurring with the types assigned to Z, and lifted by a common lifter t_Z.

$$
\frac{}{\begin{array}{l} [\![G_s]\!]^p_{\tilde{q}} = X\langle \mu_p, (p_q)_{q \in \tilde{q}} \rangle \\ \vdash \left(X : \left(\begin{array}{l} \mathsf{deepUnfold}(\overline{G_X \lfloor^{\mathsf{o}_X} p}, (Y, t_Y, \overline{G_Y \lfloor^{\mathsf{o}_Y} p})_{Y \in \widetilde{Y_X}}), \\ \left(\mathsf{deepUnfold}(\llparenthesis G_X \upharpoonright (p,q) \rrparenthesis^{\mathsf{o}_X}_{p \rangle q}, (Y, t_Y, \llparenthesis G_Y \upharpoonright (p,q) \rrparenthesis^{\mathsf{o}_Y}_{p \rangle q})_{Y \in \widetilde{Y_X}} \right)_{q \in \tilde{q}_X} \end{array} \right) \right)_{X \in \widetilde{X_C} \setminus (Z)}, \\ \quad Z : \left(\begin{array}{l} \mathsf{deepUnfold}(\overline{G_Z \lfloor^{\mathsf{o}_Z} p}, (X, t_X, \overline{G_X \lfloor^{\mathsf{o}_X} p})_{X \in \widetilde{Y_Z}}), \\ \left(\mathsf{deepUnfold}(\llparenthesis G_Z \upharpoonright (p,q) \rrparenthesis^{\mathsf{o}_Z}_{p \rangle q}, (X, t_X, \llparenthesis G_X \upharpoonright (p,q) \rrparenthesis^{\mathsf{o}_X}_{p \rangle q})_{X \in \widetilde{Y_Z}} \right)_{q \in \tilde{q}} \end{array} \right); \\ \quad \mu_p : \mu Z \,.\, (\uparrow^{t_Z} \mathsf{deepUnfold}(\overline{G_Z \lfloor^{\mathsf{o}_Z} p}, (X, t_X, \overline{G_X \lfloor^{\mathsf{o}_X} p})_{X \in \widetilde{Y_Z}})), \\ \quad \left(p_q : \mu Z \,.\, (\uparrow^{t_Z} \mathsf{deepUnfold}(\llparenthesis G_Z \upharpoonright (p,q) \rrparenthesis^{\mathsf{o}_Z}_{p \rangle q}, (X, t_X, \llparenthesis G_X \upharpoonright (p,q) \rrparenthesis^{\mathsf{o}_X}_{p \rangle q})_{X \in \widetilde{Y_Z}})) \right)_{q \in \tilde{q}} \end{array}} \textsc{Var}
$$

In this typing, the type of μ_p concurs with the expected type in (27), the types of p_q for each $q \in \tilde{q}$ concur with the expected types in (28), and the assignment to Z in the recursive context concurs with (29). This proves the thesis. □

Now, we can prove Theorem 11 as a corollary of Theorem 16:

Proof of Theorem 11 on Page 20. We have been given a closed, relative well-formed global type G, and a participant $p \in \mathsf{prt}(G)$. Let $C := []$ and $G_s := G$. Clearly, $G_s \preceq_C G$. By Definition 33, $\mathsf{active}(C, G) = \mathsf{prt}(G)^2$. For p to be a participant of G, there must be an exchange involving p and some other participant q, i.e. there exists a $q \in \mathsf{prt}(G)$ such that $(p,q) \in \mathsf{active}(C, G)$. Moreover, $\tilde{q}$ as defined in Theorem 16 is $\{q \in \mathsf{prt}(G) \mid (p,q) \in \mathsf{active}(C, G)\} = q \in \mathsf{prt}(G) \setminus \{p\}$. Hence, Theorem 16 allows us to find a typing for $[\![G]\!]^{\mathsf{prt}(G)\setminus\{p\}}_p$.

Let us consider the precise values of the ingredients of Theorem 16 in our application:

1. $\mathsf{o}_C = \mathsf{ctxpri}(C) = 0$,
2. $\widetilde{X_C} = \mathsf{ctxbind}(C) = ()$,
3. $D_p = \mathsf{deepUnfold}(\overline{G_s \lfloor^{\mathsf{o}_C} p}, (X, t_X, \overline{G_X \lfloor^{\mathsf{o}_X} p})_{X \in \widetilde{X_C}})$
   $= \overline{G \lfloor^0 p}$ (cf. Definition 34),
4. $E_q = \mathsf{deepUnfold}(\llparenthesis G_s \upharpoonright (p,q) \rrparenthesis^{\mathsf{o}_C}_{p \rangle q}, (X, t_X, \llparenthesis G_X \upharpoonright (p,q) \rrparenthesis^{\mathsf{o}_X}_{p \rangle q})_{X \in \widetilde{X_C}})$
   $= \llparenthesis G \upharpoonright (p,q) \rrparenthesis^0_{p \rangle q}$ (cf. Definition 34).

Finally, the result of Theorem 16 is as follows:

$$
[\![G_s]\!]^{\tilde{q}}_p \vdash \left(X : \left(A_X, (B_{X,q})_{q \in \tilde{q}_X} \right) \right)_{X \in \widetilde{X_C}}; \ \mu_p : D_p, \ (p_q : E_q)_{q \in \tilde{q}}
$$

Applying (1)–(4) above, we get the following:

$$
[\![G]\!]^{\mathsf{prt}(G)\setminus\{p\}}_p \vdash \emptyset; \ \mu_p : \overline{G \lfloor^0 p}, \ \left(p_q : \llparenthesis G \upharpoonright (p,q) \rrparenthesis^0_{p \rangle q} \right)_{q \in \mathsf{prt}(G)\setminus\{p\}}
$$

This coincides exactly with the result of Theorem 11. □

4.3.2. Transference of results (operational correspondence)

Given a global type G, we now formalize the transference of correctness properties such as deadlock-freedom from net(G) (cf. Definition 26) to G. Here, we define an *operational correspondence* between networks and global types, in both directions. That is, we show that a network performs interactions between implementations and routers and between pairs of routers if and only if that communication step is stipulated in the corresponding global type (Theorems 19 and 23).

Before formalizing the operational correspondence, we show that networks of routed implementations never reduce to alarm processes. To be precise, because alarm processes only can occur in routers (not in implementations), we show that none of the routers of a network reduces to an alarm process, formalized using evaluation contexts (Definition 8):

Theorem 17. *Given a relative well-formed global type G and a network of routed implementations $\mathcal{N} \in \text{net}(G)$, then*

$$\mathcal{N} \not\longrightarrow^* \mathcal{E}[\mathsf{alarm}(\tilde{x})],$$

for any reduction context $\mathcal{E}$ and set of endpoints $\tilde{x}$.

Proof. By definition (Definition 26), $\mathcal{N}$ consists only of routers (Definition 20) and well-typed processes not containing the alarm process (cf. the assumption below Definition 28).

Suppose, for contradiction, that there are $\mathcal{E}$ and $\tilde{x}$ such that $\mathcal{N} \longrightarrow^* \mathcal{E}[\mathsf{alarm}(\tilde{x})]$. Since only routers can contain the alarm process, there is a router $\mathcal{R}_p$ in $\mathcal{N}$ for participant $p \in \text{prt}(G)$ that reduces to the alarm process. Since it is the only possibility for a router synthesized by Algorithm 1 to contain the alarm process, it must contain the process in (2). This process is synthesized on line 12 of Algorithm 1, so there is an exchange in G with sender $s \in \text{prt}(G) \setminus \{p\}$ and recipient $r \in \text{prt}(G) \setminus \{p\}$ that is a dependency for the interactions of p with both s and r.

For this exchange, the router $\mathcal{R}_s$ for s contains the process returned on line 5 of Algorithm 1, and the router $\mathcal{R}_r$ for r contains the process returned on line 6. Suppose s has a choice between the labels in I, and the implementation of s chooses $i \in I$. Then, $\mathcal{R}_s$ sends i to $\mathcal{R}_r$ and $\mathcal{R}_p$.

Now, for $\mathcal{R}_p$ to reduce to the alarm process, it has to receive from $\mathcal{R}_r$ a label $i' \in I \setminus \{i\}$. However, this contradicts line 6 of Algorithm 1, which clearly defines $\mathcal{R}_r$ to send i to $\mathcal{R}_p$. Hence, $\mathcal{N} \not\longrightarrow^* \mathcal{E}[\mathsf{alarm}(\tilde{x})]$. □

It follows from this and the typability of routers (Theorem 11) that networks of routed implementations are deadlock-free:

Theorem 18. *For relative well-formed global type G, every $\mathcal{N} \in \text{net}(G)$ is deadlock-free.*

Proof. By the typability of routers (Theorem 11) and the duality of the types of router channels (Theorem 9), $\mathcal{N} \vdash \emptyset; \emptyset$. Hence, by Theorem 5, $\mathcal{N}$ is deadlock-free, and by Theorem 17, $\mathcal{N}$ never reduces to the alarm process. □

To formalize our operational correspondence result, we apply the labeled reductions for processes $P \xrightarrow{\alpha} Q$ (cf. Definition 10) and define a labeled transition system (LTS) for global types.

Definition 35 *(LTS for global types).* We define the relation $G \xrightarrow{\alpha} G'$, with labels β of the form $p \rangle q : \ell\langle S \rangle$ (sender, recipient, label, and message type), by the following rules:

$$\frac{j \in I}{p \twoheadrightarrow q\{i\langle S_i\rangle \,.\, G_i\}_{i\in I} \xrightarrow{p\rangle q : j\langle S_j\rangle} G_j} \qquad \frac{G \xrightarrow{\alpha} G'}{\mathsf{skip} \,.\, G \xrightarrow{\alpha} G'} \qquad \frac{G\{\mu X \,.\, G/X\} \xrightarrow{\alpha} G'}{\mu X \,.\, G \xrightarrow{\alpha} G'}$$

Intuitively, operational correspondence states:

1. every transition of a global type is mimicked by a precise sequence of labeled reductions originating from an associated completable network (*completeness*; Theorem 19), and
2. for every labeled reduction originated in a completable network there is a corresponding global type transition (*soundness*; Theorem 23).

We write $\rho_1\rho_2$ for the composition of relations ρ_1 and ρ_2. Recall that the notation $\longrightarrow^\star$ stands for finite sequences of reductions, as defined in Notation 2.

Theorem 19 *(Operational correspondence: completeness). Suppose given a relative well-formed global type G. Also, suppose given $p, q \in \text{prt}(G)$ and a set of labels J such that $j \in J$ if and only if $G \xrightarrow{p\rangle q : j\langle S_j\rangle} G_j$ for some S_j. Then,*

1. *for any completable* $\mathcal{N} \in \text{net}(G)$*, there exists a* $j' \in J$ *such that* $\mathcal{N}^{\circlearrowleft} \longrightarrow^{\star} \xrightarrow{p_\mu \rangle \mu_p : j'} \mathcal{N}_0$;
2. *for any* $j' \in J$*, there exists a completable* $\mathcal{N} \in \text{net}(G)$ *such that* $\mathcal{N}^{\circlearrowleft} \longrightarrow^{\star} \xrightarrow{p_\mu \rangle \mu_p : j'} \mathcal{N}_0$;
3. *for any completable* $\mathcal{N} \in \text{net}(G)$ *and any* $j' \in J$*, if* $\mathcal{N}^{\circlearrowleft} \longrightarrow^{\star} \xrightarrow{p_\mu \rangle \mu_p : j'} \mathcal{N}_0$*, then there exists a completable* $\mathcal{N}_{j'} \in \text{net}(G_{j'})$ *such that,*

$$\mathcal{N}_0 \xrightarrow{p_q \rangle q_p : j'} \longrightarrow^{\star} \xrightarrow{\mu_q \rangle q_\mu : j'} \longrightarrow^{\star} \xrightarrow{p_\mu \rangle \mu_p : v} \xrightarrow{p_q \rangle q_p : w} \xrightarrow{v \leftrightarrow w} \longrightarrow^{\star} \xrightarrow{\mu_q \rangle q_\mu : w} \xrightarrow{v \leftrightarrow w} \mathcal{N}_{j'}^{\circlearrowleft}.$$

Proof. By the labeled transitions of global types (Definition 35) and relative well-formedness, G is a sequence of skips followed by an exchange from p to q over the labels in J. Since the skips do not influence the behavior of routers, let us assume simply that

$$G = p \twoheadrightarrow q\{j\langle S_j \rangle \,.\, G_j\}_{j \in J}.$$

We prove each Sub-item separately.

(a) Take any completable $\mathcal{N} \in \text{net}(G)$. By definition (Definition 27), $\mathcal{N}^{\circlearrowleft} \vdash \emptyset; \emptyset$. By the construction of networks of routed implementations (Definition 26), $p_\mu \in \text{bn}(\mathcal{N}^{\circlearrowleft})$, and p_μ is connected to μ_p.
Also by construction, the type of p_μ in the typing derivation of $\mathcal{N}^{\circlearrowleft}$ is

$$G \upharpoonright^0 p = \oplus^0\{j : (\!|S_j|\!) \otimes^1 (G_j \upharpoonright^4 p)\})_{j \in J}.$$

By the well-typedness of $\mathcal{N}^{\circlearrowleft}$, we can infer the kind of action that is defined on p_μ: a selection, or a forwarder. By induction on the number of connected forwarders (which is finite by the finiteness of process terms), eventually a forwarder has to be connected to a selection. So, after reducing the forwarders, we have a selection on p_μ, of some $j' \in J$.
Hence, by Fairness (Theorem 7), after a finite number of steps, we can observe a communication of the label j' from p_μ to μ_p. This proves the thesis: $\mathcal{N}^{\circlearrowleft} \longrightarrow^{\star} \xrightarrow{p_\mu \rangle \mu_p : j'} \mathcal{N}_0$.

(b) Following the proof of the existence of completable networks (Proposition 10), we can generate an implementation process for all of G's participants from local projections (cf. Proposition 1). Take any $j' \in J$. For the implementation process of p, we specifically generate an implementation process that sends the label j'. These implementation processes allow us to construct $\mathcal{N}$, which by construction is in $\text{net}(G)$ and is completable. Following the reasoning as in Subitem (a), $\mathcal{N}^{\circlearrowleft} \longrightarrow^{\star} \xrightarrow{p_\mu \rangle \mu_p : j'} \mathcal{N}_0$.

(c) By definition (Definition 27), $\mathcal{N}^{\circlearrowleft} \vdash \emptyset; \emptyset$. Hence, by Fairness (Theorem 7), for any of the pending names of $\mathcal{N}^{\circlearrowleft}$, we can observe a communication after a finite number of steps. By construction (Definition 26), the endpoints that we are required to observe by thesis are bound in $\mathcal{N}^{\circlearrowleft}$. From the shape of G, the definition of routed implementations (Definition 25), and the typability of routers (Theorem 11), we know the types of all the required endpoints in $\mathcal{N}^{\circlearrowleft}$. We can deduce the required labeled reductions following the reasoning as in Subitem (a). Let us summarize the origin of each of the network's steps:

1. $\mathcal{N}^{\circlearrowleft} \longrightarrow^{\star} \xrightarrow{p_\mu \rangle \mu_p : j'} \mathcal{N}_0$: The implementation of p selects label j' with p's router.
2. $\mathcal{N}_0 \xrightarrow{p_q \rangle q_p : j'} \mathcal{N}_1$: The router of p forwards j' to q's router.
3. $\mathcal{N}_1 \longrightarrow^{\star} \mathcal{N}_2$: The router of p forwards j' to the routers of the participant that depend on the output by p, and these routers forward j' to their respective implementations.
4. $\mathcal{N}_2 \xrightarrow{\mu_q \rangle q_\mu : j'} \longrightarrow^{\star} \mathcal{N}_3$: The router of q forwards j' to q's implementation, and to the routers of the participants that depend on the input by q, and these routers forward j' to their respective implementation (if they have not done so already for the output dependency on p).
5. $\mathcal{N}_3 \xrightarrow{p_\mu \rangle \mu_p : v} \xrightarrow{p_q \rangle q_p : w} \xrightarrow{v \leftrightarrow w} \mathcal{N}_{4,v}$: The implementation of p sends an endpoint v to p's router, which sends a fresh endpoint w to q's router, and v is forwarded to w.
6. $\mathcal{N}_{4,v} \longrightarrow^{\star} \xrightarrow{\mu_q \rangle q_\mu : w} \xrightarrow{v \leftrightarrow w} \mathcal{N}_{j'}^{\circlearrowleft}$: The router of q sends a fresh endpoint w to q's implementations, and v is forwarded to w.
In $\mathcal{N}_{j'}^{\circlearrowleft}$, all routers have transitioned to routers for $G_{j'}$. Moreover, by Type Preservation (Theorem 2), $\mathcal{N}_{j'}^{\circlearrowleft} \vdash \emptyset; \emptyset$. By isolating restrictions on endpoints that belong only to implementation processes, we can find $\mathcal{N}_{j'} \in \text{net}(G_{j'})$ such that $\mathcal{N}_{j'}^{\circlearrowleft}$ is its completion. This proves the thesis.

Note that G can also contain recursive definitions before the initial exchange; this case can be dealt with by unfolding. □

Our soundness result, given below as Theorem 23, will capture the notion that after any sequence of reductions from the network of a global type G, a network of another global type G' can be reached. Crucially, G' can be reached from G through a series of transitions. Networks are inherently concurrent, whereas global types are built out of sequential compositions; as a result, the network could have enabled (asynchronous) actions that correspond to exchanges that are not immediately enabled in the global type.

For example, consider the global types $G = a \twoheadrightarrow b\{1\langle S_1\rangle.c \twoheadrightarrow d\{1\langle S'\rangle.\mathsf{end}\}, 2\langle S_2\rangle.c \twoheadrightarrow d\{1\langle S'\rangle.\mathsf{end}\}\}$ and $G' = a \twoheadrightarrow b\{1\langle S\rangle.b \twoheadrightarrow c\{1\langle S'\rangle.\mathsf{end}\}\}$. Clearly, the initial exchange in G between a and b is not a dependency for the following exchange between c and d. The routers of c and d synthesized from G thus start with their exchange, without awaiting the initial exchange between a and b to complete. Hence, in a network of G, both exchanges in G may be enabled simultaneously. We further refer to exchanges that may be simultaneously enabled in networks as *independent (global) exchanges*. While all exchanges appearing in G are independent, the two exchanges in G' are not.

In the proof of soundness, we may encounter in network reductions related to independent exchanges, so we have to be able to identify the independent exchanges in the global type to which the network belongs. Lemma 21 states that independent exchanges related to observed reductions in a network of a global type G can be reached from G after any sequence of transitions in a finite number of steps. The proof of this lemma relies on Lemma 20, which ensures that if a participant does not depend on a certain exchange, then the routers synthesized at each of the branches of the exchange are equal.

Lemma 20. *Suppose given a relative well-formed global type $G = s \twoheadrightarrow r\{i\langle S_i\rangle.G_i\}_{i\in I}$, and take any $p \in \mathsf{prt}(G) \setminus \{s, r\}$ and $\tilde{q} \subseteq \mathsf{prt}(G) \setminus \{p\}$. If neither $\mathsf{hdep}(p, s, G)$ nor $\mathsf{hdep}(p, r, G)$ holds, then $[\![G_i]\!]^{\tilde{q}}_p = [\![G_j]\!]^{\tilde{q}}_p$ for every $i, j \in I$.*

Proof. The analysis proceeds by cases on the structure of G. As a representative case we consider $G = s \twoheadrightarrow r\{1\langle S_1\rangle.G_1, 2\langle S_2\rangle.G_2\}$. Towards a contradiction, we assume $[\![G_1]\!]^{\tilde{q}}_p \neq [\![G_2]\!]^{\tilde{q}}_p$. There are many cases where Algorithm 1 generates different routers for p at G_1 and at G_2. We discuss the interesting case where $[\![G_1]\!]^{\tilde{q}}_p = \mu_p \triangleright \ldots$ (line 5) and $[\![G_2]\!]^{\tilde{q}}_p = p_{q_2} \triangleright \ldots$ (line 6). Then $G_1 = p \twoheadrightarrow q_1\{\ldots\}$ and $G_2 = q_2 \twoheadrightarrow p\{\ldots\}$. We have $G_1 \upharpoonright (p, q_1) = p\{\ldots\}$ and $G_2 \upharpoonright (p, q_1) = \mathsf{skip}\ldots$ or $G_2 \upharpoonright (p, q_1) = p?q_2\{\ldots\}$ (w.l.o.g., assume the former). Since G is relative well-formed, the projection $G \upharpoonright (p, q_1)$ must exist. Hence, since $p \notin \{s, r\}$ and $G_1 \upharpoonright (p, q_1) \neq G_2 \upharpoonright (p, q_1)$, it must be the case that $q_1 \in \{s, r\}$—w.l.o.g., assume $q_1 = s$. Then $G \upharpoonright (p, q_1) = q_1!r\{1.p\{\ldots\}, 2.\mathsf{skip}\ldots\}$, and thus $\mathsf{hdep}(p, q_1, G) = \mathsf{hdep}(p, s, G)$ is true. This contradicts the assumption that $\mathsf{hdep}(p, s, G)$ is false. □

Lemma 21. *Suppose given a relative well-formed global type G and a completable $\mathcal{N} \in \mathsf{net}(G)$ such that $\mathcal{N}^{\circlearrowleft} \xrightarrow{c_\mu\rangle\mu_c:\ell}$, for some $c \in \mathsf{prt}(G)$. For every G' and $\beta_1, \ldots, \beta_n$ ($n \geq 0$) such that $G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_n} G'$ where c is not involved in any β_k (with $G = G'$ if $n = 0$), there exist G'', $d \in \mathsf{prt}(G)$, and $\beta'_1, \ldots, \beta'_m$ ($m \geq 0$) such that $G' \xrightarrow{\beta'_1} \ldots \xrightarrow{\beta'_m} G'' = c \twoheadrightarrow d\{i\langle S_i\rangle.G_i\}_{i\in I}$ where c is not involved in any β'_k (with $G'' = c \twoheadrightarrow d\{i\langle S_i\rangle.G_i\}_{i\in I}$ if $m = 0$).*

Proof. By induction on n (IH_1). We first observe that the behavior on μ_c in $\mathcal{N}^{\circlearrowleft}$ can only arise from the router generated for c at G, following Algorithm 1 (line 5) after finitely many passes through lines 13 (no dependency) and 19 (skip); for simplicity, assume only line 13 applies.

- Case $n = 0$. Let $x \geq 0$ denote the number of passes through line 13 to generate the router for c at G. We apply induction on x (IH_2):
  - Case $x = 0$. The router for c at G is generated through line 5, so $G = c \twoheadrightarrow d\{i\langle S_i\rangle.G_i\}_{i\in I}$, proving the thesis.
  - Case $x = x' + 1$. Then $G = a \twoheadrightarrow b\{i\langle S_i\rangle.G_i\}_{i\in I}$ and line 13 returns the router for c at G_j for any $j \in I$. We have $G \xrightarrow{a\rangle b:j\langle S_j\rangle} G_j$. Given the same implementation process for c as in $\mathcal{N}$, we can construct a completable $\mathcal{M} \in \mathsf{net}(G_j)$ such that $\mathcal{M}^{\circlearrowleft} \xrightarrow{c_\mu\rangle\mu_c:\ell}$. Hence, the thesis follows from IH_2.
- Case $n = n' + 1$. By assumption, $G \xrightarrow{\beta_1} G'_1$ where c is not the sender or recipient in β_1. Hence, $G = a \twoheadrightarrow b\{i.\langle S_i\rangle.G_i\}_{i\in I}$ where $G'_1 = G_j$ for some $j \in I$. The router for c at G is thus generated through line 13 of Algorithm 1. It follows from Lemma 20 that this router is equal to the router for c at G'_1, but with one less pass through line 13. Given the same implementation process for c as in $\mathcal{N}$, we can construct a completable $\mathcal{M} \in \mathsf{net}(G'_1)$ such that $\mathcal{M}^{\circlearrowleft} \xrightarrow{c_\mu\rangle\mu_c:\ell}$. Hence, the thesis follows from IH_1. □

The proof of soundness relies on Proposition 22: if different reductions are enabled for a given process, then they do not exclude each other. That is, the same process is reached no matter the order in which those reductions are executed. We refer to simultaneously enabled reductions as *independent reductions*.

Proposition 22 *(Independent reductions). Suppose given a process $P \vdash \Omega; \Gamma$ and reduction labels α and $\alpha'_1, \ldots, \alpha'_n$ $(n \geq 1)$ where $\alpha \notin \{\alpha'_1, \ldots, \alpha'_n\}$ (cf. Definition 10). If $P \xrightarrow{\alpha}$ and $P \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_n}$, then there exists a process Q such that $P \xrightarrow{\alpha} \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_n} Q$ and $P \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_n} \xrightarrow{\alpha} Q$.*

Proof. By induction on n:

- $n = 1$. By assumption, $P \xrightarrow{\alpha}$ and $P \xrightarrow{\alpha'_1}$. The proof proceeds by considering all possible combinations of shapes for α and α'_1 (forwarder, output/input, and selection/branching).
  Consider the case where $\alpha = x\rangle y : a$ and $\alpha'_1 = w\rangle z : b$. Because P is well-typed, we infer that there are evaluation contexts E_1 and E_2 such that $P \equiv E_1[(\boldsymbol{\nu} xy)(x[a, c] \mid y(a, c).P_1)] \equiv E_2[(\boldsymbol{\nu} wz)(w[b, d] \mid z(b, d).P_2])$ (Definition 8). Since the reductions labeled α and α'_1 are both enabled in P, it cannot be the case that $x, y \in \mathrm{fn}(P_2)$ and $w, z \in \mathrm{fn}(P_1)$. Hence, there exists an evaluation context E_3 such that $P \equiv E_3[(\boldsymbol{\nu} xy)(x[a, c] \mid y(a, c).P_1) \mid (\boldsymbol{\nu} wz)(w[b, d] \mid z(b, d).P_2])$. Then $P \xrightarrow{\alpha} Q_1 \equiv E_3[P_1 \mid (\boldsymbol{\nu} wz)(w[b, d] \mid z(b, d).P_2])$ and $P \xrightarrow{\alpha'_1} Q_2 \equiv E_3[(\boldsymbol{\nu} xy)(x[a, c] \mid y(a, c).P_1) \mid P_2]$. Let $Q = E_3[P_1 \mid P_2]$; then $Q_1 \xrightarrow{\alpha'_1} Q$ and $Q_2 \xrightarrow{\alpha} Q$. Hence, $P \xrightarrow{\alpha} \xrightarrow{\alpha'_1} Q$ and $P \xrightarrow{\alpha'_1} \xrightarrow{\alpha} Q$.
  All other cases proceed similarly. Note that when one of the reductions (say, α) has a selection/branching label, such a reduction would discard some branches and thus possible behaviors. This is not an issue for establishing the thesis, because typability guarantees that the sub-process that enables the α'-labeled reduction does not appear under the to-be-discarded branches. Hence, the execution of α will not jeopardize the α'-labeled reduction.
- $n = n' + 1$ for $n' \geq 1$. By the IH, $P \xrightarrow{\alpha} \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_{n'}} Q'_1$ and $P \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_{n'}} P' \xrightarrow{\alpha} Q'_1$. By assumption, $P' \xrightarrow{\alpha'_n} Q'_2$. Since P is well-typed, by Theorem 2 (Subject Reduction), P' is well-typed. Since $P' \xrightarrow{\alpha} Q'_1$ and $P' \xrightarrow{\alpha'_n} Q'_2$, we can follow the same argumentation as in the base case to show that $Q'_1 \xrightarrow{\alpha'_n} Q$ and $Q'_2 \xrightarrow{\alpha} Q$. Hence, $P \xrightarrow{\alpha} \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_{n'}} Q'_1 \xrightarrow{\alpha'_n} Q$ and $P \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_{n'}} P' \xrightarrow{\alpha'_n} Q'_2 \xrightarrow{\alpha} Q$. □

To understand the proof of soundness and the rôle of independent reductions therein, consider the following example. We first introduce some notation which we also use in the proof of soundness: given an ordered sequence of reduction labels $A = (\alpha_1, \ldots, \alpha_k)$, we write $P \xrightarrow{A} Q$ to denote $P \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_k} Q$.

Example 9. The recursive global type $G = \mu X.a \twoheadrightarrow b : 1\langle S\rangle.c \twoheadrightarrow d : 1\langle S\rangle.X$ features two independent exchanges. Consider a network $\mathcal{N} \in \mathrm{net}(G)$. Let A denote the sequence of labeled reductions necessary to complete the exchange in G between a and b, and C similarly for the exchange between c and d. Assuming that communication with routers is not blocked by implementation processes, we have $\mathcal{N}^{\circlearrowleft} \xrightarrow{A}$ and $\mathcal{N}^{\circlearrowleft} \xrightarrow{C}$, because the exchanges are independent.

Now, suppose that from $\mathcal{N}^{\circlearrowleft}$ we observe m times the sequence of C reductions: $\mathcal{N}^{\circlearrowleft} \underbrace{\xrightarrow{C} \ldots \xrightarrow{C}}_{m \text{ times}} N'$. We see that N' is not a network of a global type reachable from G: there are still m exchanges between a and b pending. Still, we can exhibit a series of transitions from G that includes m times the exchange between c and d:

$$G \underbrace{\xrightarrow{a\rangle b:1\langle S\rangle} \xrightarrow{c\rangle d:1\langle S\rangle} \ldots \xrightarrow{a\rangle b:1\langle S\rangle} \xrightarrow{c\rangle d:1\langle S\rangle}}_{m \text{ times}} G$$

Following these transitions, we can exhibit a corresponding sequence of reductions from $\mathcal{N}^{\circlearrowleft}$ that includes m times the sequence C and ends up in another network $\mathcal{M} \in \mathrm{net}(G)$:

$$\mathcal{N}^{\circlearrowleft} \underbrace{\xrightarrow{A} \xrightarrow{C} \ldots \xrightarrow{A} \xrightarrow{C}}_{m \text{ times}} \mathcal{M}^{\circlearrowleft}$$

At this point it is crucial that from $\mathcal{N}^{\circlearrowleft}$ the sequences of reductions A and C can be performed independently. Hence, by Proposition 22, $\mathcal{N}^{\circlearrowleft} \underbrace{\xrightarrow{C} \ldots \xrightarrow{C}}_{m \text{ times}} N' \underbrace{\xrightarrow{A} \ldots \xrightarrow{A}}_{m \text{ times}} \mathcal{M}^{\circlearrowleft}$.

In the proof of soundness, whenever we assure that certain reductions are independent, we refer to those assurances as *independence facts* (**IFacts**). Also, in the proof we consider labeled reductions, and distinguish between *protocol* and *implementation* reductions: the former are reductions with labels that indicate any interaction with a router, and the latter are any other reductions (which, by the definition of networks, can only occur within participant implementation processes).

By a slight abuse of notation, given ordered sequences of reduction labels A and A', we write $A' \subseteq A$ to denote that A' is a subsequence of A, where the labels in A' appear in the same order in A but not necessarily in sequence (and similarly for $A' \subset A$). With $A \setminus A'$ we denote the sequence obtained from A by removing all the labels in A', and $A \cup A'$ denotes the sequence obtained by adding the labels from A' to the end of A.

Theorem 23 *(Operational correspondence: soundness). Suppose given a relative well-formed global type G and a completable $\mathcal{N} \in \text{net}(G)$. For every ordered sequence of $k \geq 0$ reduction labels $A = (\alpha_1, \ldots, \alpha_k)$ and N' such that $\mathcal{N}^{\circlearrowleft} \xrightarrow{A} N'$, there exist G' and $\beta_1, \ldots, \beta_n$ (with $n \geq 0$) such that (i) $G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_n} G'$ and (ii) $N' \longrightarrow^* \mathcal{M}^{\circlearrowleft}$, with $\mathcal{M} \in \text{net}(G')$.*

Proof. By induction on the structure of G; we detail the interesting cases of labeled exchanges with implicitly unfolded recursive definitions. We exhibit transitions $G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_n} G'$ and establish a corresponding sequence of reductions $\mathcal{N}^{\circlearrowleft} \longrightarrow^* \mathcal{M}^{\circlearrowleft}$ that includes all the labels in A, with $\mathcal{M} \in \text{net}(G')$. During this step, we assure the independence between the observed reductions A and the reductions we establish (**IFacts**). Using these independence assurances, we show that also $N' \longrightarrow^* \mathcal{M}^{\circlearrowleft}$.

We apply induction on the size of A (IH_1) to show the existence of (i) G' and $\beta_1, \ldots, \beta_n$ such that (i) $G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_n} G'$ and (ii) $\mathcal{N}^{\circlearrowleft} \longrightarrow^* \mathcal{M}^{\circlearrowleft}$ including all reductions in A, with $\mathcal{M} \in \text{net}(G')$:

- Base case: then A is empty, and the thesis holds trivially, with $G' = G$ and $\mathcal{M} = \mathcal{N}$.
- Inductive case: then A is non-empty.
  By the definition of networks (Definition 26), we know that reductions starting at $\mathcal{N}^{\circlearrowleft}$ are protocol reductions related to an independent exchange in G, or implementation reductions. Every protocol reduction in A is related to some exchange in G, and so we can group sequences of protocol reductions related to the same exchange. By construction, every such sequence of protocol reductions $A_* \subseteq A$ starts with an implementation sending a label to a router, i.e., with a label of the form $\alpha_* = c_\mu \rangle \mu_c : \ell$. For each such α_*, the router in $\mathcal{N}$ of the sender c has been synthesized from G in a finite number of inductive steps. We take the α_* that originates from the router synthesized in the least number of steps. This gives us the A_* starting with α_* that relates to an exchange in G which is not prefixed by exchanges relating to any of the other $A'_* \subseteq A \setminus A_*$.
  Networks are well-typed by definition. None of the reductions in A_* are blocked by protocol reductions appearing earlier in A (**IFact 1**): they originate from exchanges in G appearing after the exchange related to A_*, and the priorities in their related types are thus higher than those in the types related to A_*, i.e., blocking by input or branching would contradict the well-typedness of $\mathcal{N}^{\circlearrowleft}$. However, it may be that some implementation reductions $A_+ \subseteq A \setminus A_*$ do block the reductions in A_*; they are also not blocked by any prior protocol reductions due to priorities (**IFact 2**). Hence, from $\mathcal{N}^{\circlearrowleft}$ we can perform the implementation reductions in A_+. By Subject Reduction (Theorem 2), this results in another completed network $\mathcal{N}_0^{\circlearrowleft}$ of G. This establishes the reduction sequence $\mathcal{N}^{\circlearrowleft} \xrightarrow{A_+} \mathcal{N}_0^{\circlearrowleft} \xrightarrow{A_*}$.

  By Lemma 21, there are $m \geq 0$ transitions $G \xrightarrow{\beta_1} G_1 \ldots \xrightarrow{\beta_m} G_m$ where the initial prefix of G_m corresponds to the labeled choice by the implementation of c: $G_m = c \twoheadrightarrow d\{i\langle S_i\rangle.G'_i\}_{i \in I}$, with $\ell \in I$. Additionally, G_m contains exchanges related to every sequence of protocol reductions in $A \setminus A_+ \setminus A_*$: all these sequences start with a selection from implementation to router, and thus the involved participants do not depend on any of the exchanges between G and G_m, such that Lemma 20 applies. To establish a sequence of reductions from $\mathcal{N}_0^{\circlearrowleft}$ to the completion of a network $\mathcal{N}_m \in \text{net}(G_m)$, we apply induction on m (IH_2):
  - The base case where $m = 0$ is trivial, with $G_m = G$ and thus $\mathcal{N}_m^{\circlearrowleft} = \mathcal{N}_0^{\circlearrowleft}$.
  - In the inductive case, following the same approach as in the proof of completeness (Theorem 19), we reduce $\mathcal{N}_0^{\circlearrowleft} \longrightarrow^* \mathcal{N}_1^{\circlearrowleft}$ such that $\mathcal{N}_1 \in \text{net}(G_1)$. Then, by IH_2, $\mathcal{N}_1^{\circlearrowleft} \longrightarrow^* \mathcal{N}_m^{\circlearrowleft}$ where $\mathcal{N}_m \in \text{net}(G_m)$. Note that these reductions may require implementation reductions to unblock protocol reductions, and these implementation reductions may appear in A. None of the reductions from $\mathcal{N}_0^{\circlearrowleft}$ to $\mathcal{N}_m^{\circlearrowleft}$ can be blocked by any of the other protocol reductions in A, following again from priorities in types; hence, the leftover reductions in A are independent from these reductions (**IFact 3**). Additionally, the sequence of protocol reductions A_* was already enabled from $\mathcal{N}_0^{\circlearrowleft}$, so those reductions are also independent (**IFact 4**).

  We know that $\mathcal{N}_m^{\circlearrowleft} \xrightarrow{c_\mu \rangle \mu_c : \ell}$ and $G_m \xrightarrow{c \rangle d : \ell \langle S_\ell \rangle} G'_\ell$. From $\mathcal{N}_m^{\circlearrowleft}$, we again follow the proof of completeness to show that $\mathcal{N}_m^{\circlearrowleft} \longrightarrow^* \mathcal{M}_\ell^{\circlearrowleft}$, where $\mathcal{M}_\ell \in \text{net}(G'_\ell)$. Given the definition of routers, it must be that all the reductions in A_* appear in this sequence of reductions. Let $A' \subset A$ denote the leftover reductions from A (i.e., A except all reductions that occurred between $\mathcal{N}^{\circlearrowleft}$ and $\mathcal{M}_\ell^{\circlearrowleft}$, including A_* and A_+). By **IFacts 1–4**, $\mathcal{M}_\ell^{\circlearrowleft} \xrightarrow{A'} M'$. Then by IH_1, there exist G' and $\beta_{m+2}, \ldots, \beta_n$ (with $n \geq m+1$) such that (i) $G'_\ell \xrightarrow{\beta_{m+2}} \ldots \xrightarrow{\beta_n} G'$ and (ii) $\mathcal{M}_\ell^{\circlearrowleft} \longrightarrow^* \mathcal{M}^{\circlearrowleft}$ including all reductions in A', with $\mathcal{M} \in \text{net}(G')$. Let $\beta_{m+1} = c \rangle d : \ell \langle S_\ell \rangle$. We have shown the existence of G' and $\beta_1, \ldots, \beta_n$ such that (i) $G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_m} G_m \xrightarrow{\beta_{m+1}} G'_\ell \xrightarrow{\beta_{m+2}} \ldots \xrightarrow{\beta_n} G'$ and (ii) $\mathcal{N}^{\circlearrowleft} \longrightarrow^* \mathcal{N}_m^{\circlearrowleft} \longrightarrow^* \mathcal{M}_\ell^{\circlearrowleft} \longrightarrow^* \mathcal{M}^{\circlearrowleft}$ including all reductions in A, with $\mathcal{M} \in \text{net}(G')$.

Algorithm 2: Synthesis of Orchestrator Processes (Definition 36).

1 **def** $\mathsf{O}_{\tilde{q}}[G]$ **as**
2 **switch** G **do**
3 **case** $s \twoheadrightarrow r\{i\langle S_i\rangle . G_i\}_{i\in I}$ **do**
4 deps $:= \{q \in \tilde{q} \mid \mathrm{hdep}(q,s,G) \vee \mathrm{hdep}(q,r,G)\}$
5 **return** $\mu_s \triangleright \{i : \overline{\mu_r} \triangleleft i \cdot (\underline{\overline{\mu_q} \triangleleft i})_{q\in\mathsf{deps}} \cdot \mu_s(v) \cdot \overline{\mu_r}[w] \cdot (v \leftrightarrow w \mid \mathsf{O}_{\tilde{q}}[G_i])\}_{i\in I}$
6 **case** $\mu X . G'$ **do**
7 $\tilde{q}' := \{q \in \tilde{q} \mid G \downarrow^0 q \neq \bullet\}$
8 **if** $\tilde{q}' \neq \emptyset$ **then return** $\mu X((\mu_q)_{q\in\tilde{q}'}) . \mathsf{O}_{\tilde{q}'}[G']$
9 **else return 0**
10 **case** X **do return** $X\langle(\mu_q)_{q\in\tilde{q}}\rangle$
11 **case** *skip* $. G'$ **do return** $\mathsf{O}_{\tilde{q}}[G']$
12 **case** end **do return 0**

We are left to show that from $\mathcal{N}^{\circlearrowleft} \longrightarrow^* \mathcal{M}^{\circlearrowleft}$ and $\mathcal{N}^{\circlearrowleft} \xrightarrow{A} N'$, we can conclude that $N' \longrightarrow^* \mathcal{M}^{\circlearrowleft}$. We apply induction on the size of A (IH_3), using **IFacts 1–4** and Proposition 22:

- Base case: Then A is empty, there is nothing to do, and the thesis is proven.
- Inductive case: Then $A = A' \cup (\alpha')$. By IH_3, $\mathcal{N}^{\circlearrowleft} \xrightarrow{A'} N'' \longrightarrow^* N''' \xrightarrow{\alpha'} \longrightarrow^* \mathcal{M}^{\circlearrowleft}$. Moreover, by assumption, $N'' \xrightarrow{\alpha'} N'$. **IFacts 1–4** show that the α'-labeled reduction is independent from the reductions between N'' and N'''. Hence, by Proposition 22, we have $\mathcal{N}^{\circlearrowleft} \xrightarrow{A'} N'' \xrightarrow{\alpha'} N' \longrightarrow^* \mathcal{M}^{\circlearrowleft}$. That is, $\mathcal{N}^{\circlearrowleft} \xrightarrow{A} N' \longrightarrow^* \mathcal{M}^{\circlearrowleft}$, proving the thesis. □

In the light of Theorem 23, let us revisit Example 9:

Example 10 *(Revisiting Example 9).* Recall the global type $G = \mu X.a \twoheadrightarrow b : 1\langle S\rangle.c \twoheadrightarrow d : 1\langle S\rangle.X$ from Example 9, with two independent exchanges. We take some $\mathcal{N} \in \mathrm{net}(G)$ such that $\mathcal{N}^{\circlearrowleft} \underbrace{\xrightarrow{C} \ldots \xrightarrow{C}}_{m \text{ times}} N'$, where C denotes the sequence of reduction labels corresponding to the exchange between c and d. By Theorem 23, there indeed are G' and $\beta_1, \ldots, \beta_n$ such that $G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_n} G'$ and $N' \longrightarrow^* \mathcal{M}^{\circlearrowleft}$, with $\mathcal{M} \in \mathrm{net}(G')$. To be precise, following Theorem 23, indeed

$$G \underbrace{\xrightarrow{a\rangle b:1\langle S\rangle} \xrightarrow{c\rangle d:1\langle S\rangle} \ldots \xrightarrow{a\rangle b:1\langle S\rangle} \xrightarrow{c\rangle d:1\langle S\rangle}}_{m \text{ times}} G \quad \text{and} \quad \mathcal{N}^{\circlearrowleft} \underbrace{\xrightarrow{C} \ldots \xrightarrow{C}}_{m \text{ times}} N' \underbrace{\xrightarrow{A} \ldots \xrightarrow{A}}_{m \text{ times}} \mathcal{M}^{\circlearrowleft}$$

where A is the sequence of reduction labels corresponding to the exchange between a and b and $\mathcal{M} \in \mathrm{net}(G)$.

4.4. Routers strictly generalize centralized orchestrators

Unlike our decentralized analysis, previous analyses of global types using binary session types rely on centralized orchestrators (called mediums [12] or arbiters [17]). Here, we show that our approach strictly generalizes these centralized approaches. Readers interested in our decentralized approach in action may safely skip this section and go directly to Section 5.

We introduce an algorithm that synthesizes an orchestrator—a single process that orchestrates the interactions between a protocol's participants (§4.4.1). We show that the composition of this orchestrator with a context of participant implementations is behaviorally equivalent to the specific case in which routed implementations are organized in a *centralized composition* (Theorem 27 in §4.4.2).

4.4.1. Synthesis of orchestrators

We define the synthesis of an orchestrator from a global type. The orchestrator of G will have a channel endpoint μ_{p_i} for connecting to the process implementation of every $p_i \in \mathsf{prt}(G)$.

Definition 36 *(Orchestrator).* Given a global type G and participants $\tilde{q}$, Algorithm 2 defines the synthesis of an *orchestrator process*, denoted $\mathsf{O}_{\tilde{q}}[G]$, that orchestrates interactions according to G.

Algorithm 2 follows a similar structure as the router synthesis algorithm (Algorithm 1). The input parameter $\tilde{q}$ keeps track of active participants, making sure recursions are well-defined; it should be initialized as $\mathsf{prt}(G)$.

We briefly discuss how the orchestrator process is generated. The interesting case is an exchange $p \twoheadrightarrow q\{i\langle U_i\rangle . G_i\}_{i\in I}$ (line 3), where the algorithm combines the several cases of the router's algorithm (that depend on the involvement of the router's participant). First, the sets of participants deps that depend on the sender and on the recipient are computed

(line 4) using the auxiliary predicate hdep (cf. Definition 19). Then, the algorithm returns a process (line 5) that receives a label $i \in I$ over μ_s; forwards it over μ_r and over μ_q for all $q \in \mathsf{deps}$; receives a channel over μ_s; forwards it over μ_r; and continues as $\mathsf{O}_{\tilde{q}}[G_i]$.

The synthesis of a recursive definition $\mu X \,.\, G'$ (line 6) requires care, as the set of active participants $\tilde{q}$ may change. In order to decide which $q \in \tilde{q}$ are active in G', the algorithm computes the local projection of G onto each $q \in \tilde{q}$ to determine the orchestrator's future behavior on μ_q, creating a new set $\tilde{q}'$ with those $q \in \tilde{q}$ for which the projection is different from $\bullet$ (line 7). Then, the algorithm returns a recursive process with as context the channel endpoints μ_q for $q \in \tilde{q}'$, with $\mathsf{O}_{\tilde{q}'}[G']$ as the body.

The synthesis of a recursive call X (line 10) yields a recursive call with as context the channels μ_q for $q \in \tilde{q}$. Finally, for $\mathsf{skip} \,.\, G'$ (line 11) the algorithm returns the orchestrator for G', and for $\bullet$ (line 12) the algorithm returns $\mathbf{0}$.

There is a minor difference between the orchestrators synthesized by Algorithm 2 and the mediums defined by Caires and Pérez [12]. The difference is in the underlined portion in line 5, which denotes explicit messages (obtained via dependency detection) needed to deal with non-local choices. The mediums by Caires and Pérez do not include such communications, as their typability is based on local types, which rely on a merge operation at projection time. The explicit actions in line 5 make the orchestrator compatible with participant implementations that connect with routers. Aside from these actions, our concept of orchestrator is essentially the same as that of mediums.

Crucially, orchestrators can be typed using local projection (cf. Definition 23) similar to the typing of routers using relative projection (cf. Theorem 11). This result follows by construction:

Theorem 24. *Given a closed, relative well-formed global type G,*

$$\mathsf{O}_{\mathsf{prt}(G)}[G] \vdash \emptyset;\ (\mu_p : \overline{(G \restriction^0 p)})_{p \in \mathsf{prt}(G)}.$$

Proof. We prove a more general statement. Suppose given a closed, relative well-formed global type G. Also, suppose given a global type $G_s \preceq_C G$. Consider:

- the participants that are active in G_s: $\tilde{q} = \{q \in \mathsf{prt}(G) \mid \exists p \in \mathsf{prt}(G).\ (p, q) \in \mathsf{active}(C, G)\}$,
- the absolute priority of G_s: $\mathsf{o}_C = \mathsf{ctxpri}(C)$,
- the sequence of bound recursion variables of G_s: $\widetilde{X_C} = \mathsf{ctxbind}(C)$,
- for every $X \in \widetilde{X_C}$:
  - the body of the recursive definition on X in G: $G_X = \mathsf{recdef}(X, G)$,
  - the participants that are active in G_X: $\tilde{q}_X = \{q \in \mathsf{prt}(G) \mid \exists p \in \mathsf{prt}(G).\ (p, q) \in \mathsf{recactive}(X, G)\}$,
  - the absolute priority of G_X: $\mathsf{o}_X = \mathsf{varpri}(X, G)$,
  - the sequence of bound recursion variables of G_X excluding X: $\widetilde{Y_X} = \mathsf{subbind}(\mu X \,.\, G_X, G)$,
  - the type required for μ_q for a recursive call on X:

$$A_{X,q} = \mathsf{deepUnfold}(\overline{G_X \restriction^{\mathsf{o}_X} q}, (Y, t_Y, \overline{G_Y \restriction^{\mathsf{o}_Y} q})_{Y \in \widetilde{Y_X}})$$

  - the minimum lift for typing a recursive definition on X: $t_X = \max_{\mathsf{pr}}\left((A_{X,q})_{q \in \tilde{q}_X}\right) + 1$,
- the type expected for μ_q for the orchestrator for G_s:

$$D_q = \mathsf{deepUnfold}(\overline{G_s \restriction^{\mathsf{o}_C} q}, (X, t_X, \overline{G_X \restriction^{\mathsf{o}_X} q})_{X \in \widetilde{X_C}}).$$

Then, we have:

$$\mathsf{O}_{\tilde{q}}[G_s] \vdash \left(X : (A_{X,q})_{q \in \tilde{q}_X}\right)_{X \in \widetilde{X_C}};\ (\mu_q : D_q)_{q \in \tilde{q}}$$

Similar to how Theorem 11 follows from Theorem 16, the thesis follows as a corollary from this more general statement (cf. the proof of Theorem 11 on Page 30).

We apply induction on the structure of G_s, with six cases as in Algorithm 2. We only detail the cases of exchange and recursion.

- *Exchange*: $G_s = s \twoheadrightarrow r\{i\langle S_i\rangle \,.\, G_i\}_{i \in I}$ (line 3).
  Following similar reasoning as in the case for exchange in the proof of Theorem 16, we can omit the unfoldings on types, as well as the recursive context.
  Let $\mathsf{deps}_s := \{q \in \tilde{q} \mid \mathsf{hdep}(q, s, G_s)\}$ and $\mathsf{deps}_r := \{q \in \tilde{q} \setminus \mathsf{deps}_s \mid \mathsf{hdep}(q, r, G_s)\}$. Note that $\mathsf{deps}_s \cup \mathsf{deps}_r$ coincides with deps as defined on line 4 and that $s, r \notin \mathsf{deps}_s \cup \mathsf{deps}_r$.
  Let us take stock of the types we expect for each of the orchestrator's channels.

  For μ_s we expect $\overline{G_s \restriction^{\mathsf{o}} s} = \overline{\oplus^{\mathsf{o}}\{i : (\!|S_i|\!) \otimes^{\mathsf{o}+1} (G_i \restriction^{\mathsf{o}+4} s)\}_{i \in I}}$

$$= \&^{o}\{i : \overline{(\!|S_i|\!)} \parr^{o+1} \overline{(G_i \lfloor^{o+4} s)}\}_{i\in I}. \tag{30}$$

For μ_r we expect
$$\overline{G_s \lfloor^{o} r} = \overline{\&^{o+2}\{i : \overline{(\!|S_i|\!)} \parr^{o+3} (G_i \lfloor^{o+4} r)\}_{i\in I}}$$
$$= \oplus^{o+2}\{i : (\!|S_i|\!) \otimes^{o+3} \overline{(G_i \lfloor^{o+4} r)}\}_{i\in I}. \tag{31}$$

For each $q \in \mathsf{deps}_s$,

for μ_q we expect
$$\overline{G_s \lfloor^{o} q} = \overline{\&^{o+2}\{i : (G_i \lfloor^{o+4} q)\}_{i\in I}}$$
$$= \oplus^{o+2}\{i : \overline{(G_i \lfloor^{o+4} q)}\}_{i\in I}. \tag{32}$$

For each $q \in \mathsf{deps}_r$,

for μ_q we expect
$$\overline{G_s \lfloor^{o} q} = \overline{\&^{o+3}\{i : (G_i \lfloor^{o+4} q)\}_{i\in I}}$$
$$= \oplus^{o+3}\{i : \overline{(G_i \lfloor^{o+4} q)}\}_{i\in I}. \tag{33}$$

For each $q \in \tilde{q} \setminus \mathsf{deps}_s \setminus \mathsf{deps}_r \setminus \{s, r\}$,

for μ_q we expect
$$\overline{G_s \lfloor^{o} q} = \overline{G_{i'} \lfloor^{o+4} q} \text{ for any } i' \in I. \tag{34}$$

Let us now consider the process returned by Algorithm 1, with each prefix marked with a number.

$$\mathsf{O}_{\tilde{q}}[G] = \underbrace{\mu_s \triangleright \{i :}_{1} \underbrace{\overline{\mu_r} \triangleleft i}_{2_i} \cdot \underbrace{(\overline{\mu_q} \triangleleft i)_{q\in\mathsf{deps}}}_{3_i} \cdot \underbrace{\mu_s(v)}_{4_i} . \underbrace{\overline{\mu_r}[w]}_{5_i} \cdot (v \leftrightarrow w \mid \mathsf{O}_{\tilde{q}}[G_i])\}_{i\in I}$$

For each $i' \in I$, let $C_{i'} := C[s \twoheadrightarrow r(\{i\langle S_i\rangle \,.\, G_i\}_{i\in I\setminus\{i'\}} \cup \{i'\langle S_{i'}\rangle \,.\, []\})]$. Clearly, $G_{i'} \leq_{C_{i'}} G$. Also, because we are not adding recursion binders, the current value of $\tilde{q}$ is appropriate for the IH. With $C_{i'}$ and $\tilde{q}$, we apply the IH to obtain the typing of $\mathsf{O}_{\tilde{q}}[G_{i'}]$, where priorities start at $\mathrm{ctxpri}(C_{i'}) = \mathrm{ctxpri}(C) + 4$ (cf. Definition 32). Following these typings, Fig. 12 gives the typing of $\mathsf{O}_{\tilde{q}}[G_s]$, referring to parts of the process by the number marking its foremost prefix above.
Clearly, the priorities in the derivation in Fig. 12 meet all requirements. The order of the applications of $\oplus^\star$ for each $q \in \mathsf{deps}_s \cup \mathsf{deps}_r$ does not matter, since the selection actions are asynchronous.

- *Recursive definition*: $G_s = \mu Z \,.\, G'$ (line 6). Let

$$\tilde{q}' := \{q \in \tilde{q} \mid G_s \lfloor^{o} q \neq \bullet\} \tag{35}$$

(as on line 7). The analysis depends on whether $\tilde{q}' = \emptyset$ or not.

– If $\tilde{q}' = \emptyset$ (line 9), let us take stock of the types expected for each of the orchestrator's channels. For now, we omit the substitutions in the types.

For each $q \in \tilde{q}$, for μ_q we expect
$$\overline{G_s \lfloor^{o_C} q} = \bullet. \tag{36}$$

Because all expected types are $\bullet$, the substitutions do not affect the types, so we can omit them altogether.
First we apply Empty, giving us an arbitrary recursive context, thus the recursive context we need. Then, we apply $\bullet$ for μ_q for each $q \in \tilde{q}$ (cf. (36)), and obtain the typing of $\mathsf{O}_{\tilde{q}}[G_s]$ (omitting the recursive context):

$$\mathsf{O}_{\tilde{q}}[G_s] = \mathbf{0} \vdash (\mu_q : \bullet)_{q\in\tilde{q}}.$$

– If $\tilde{q}' \neq \emptyset$ (line 8), let us take stock of the types expected for each of the orchestrator's channels. Note that, because of the recursive definition on Z in G_s, there cannot be another recursive definition in the context C capturing the recursion variable Z. Therefore, by Definition 30, $Z \notin \widetilde{X_C}$.

For each $q \in \tilde{q}'$,

for μ_q we expect
$$\mathrm{deepUnfold}(\overline{G_s \lfloor^{o_C} q}, \ldots)$$
$$= \mathrm{deepUnfold}(\overline{\mu Z \,.\, (G' \lfloor^{o_C} q)}, \ldots)$$
$$= \mathrm{deepUnfold}(\mu Z \,.\, \overline{G' \lfloor^{o_C} q}, \ldots)$$
$$= \mu Z \,.\, \mathrm{deepUnfold}(\overline{G' \lfloor^{o_C} q}, (X, t_X, \overline{G_X \lfloor^{o_X} q})_{X\in\widetilde{X_C}}). \tag{37}$$

For each $q \in \tilde{q} \setminus \tilde{q}'$,

for μ_q we expect
$$\mathrm{deepUnfold}(\overline{G_s \lfloor^{o_C} q}, \ldots)$$
$$= \mathrm{deepUnfold}(\bullet, \ldots) = \bullet. \tag{38}$$

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\forall i \in I.\ v \leftrightarrow w \vdash v : \overline{S_i},\ w : S_i}\ \textsc{Id}
\qquad
\forall i \in I.\ \mathsf{O}_{\tilde{q}}[G_i] \vdash (\mu_q : \overline{G_i \lfloor^{\mathsf{o}+4} q})_{q \in \tilde{q}}
}{\forall i \in I.\ v \leftrightarrow w \mid \mathsf{O}_{\tilde{q}}[G_i] \vdash v : \overline{S_i},\ w : S_i,\ (\mu_q : \overline{G_i \lfloor^{\mathsf{o}+4} q})_{q \in \tilde{q}}}\ \textsc{Mix}
}{\forall i \in I.\ 5_i \vdash v : \overline{S_i},\ \mu_r : (\!|S_i|\!) \otimes^{\mathsf{o}+3} \overline{(G_i \lfloor^{\mathsf{o}+4} r)},\ (\mu_q : \overline{G_i \lfloor^{\mathsf{o}+4} q})_{q \in \tilde{q} \setminus \{r\}}}\ \otimes^\star
}{\forall i \in I.\ 4_i \vdash \mu_s : \overline{(\!|S_i|\!)} \parr^{\mathsf{o}+1} \overline{(G_i \lfloor^{\mathsf{o}+4} s)},\ \mu_r : (\!|S_i|\!) \otimes^{\mathsf{o}+3} \overline{(G_i \lfloor^{\mathsf{o}+4} r)},\ (\mu_q : \overline{G_i \lfloor^{\mathsf{o}+4} q})_{q \in \tilde{q} \setminus \{s,r\}}}\ \parr
}{\forall i \in I.\ 3_i \vdash \mu_s : \overline{(\!|S_i|\!)} \parr^{\mathsf{o}+1} \overline{(G_i \lfloor^{\mathsf{o}+4} s)},\ \mu_r : (\!|S_i|\!) \otimes^{\mathsf{o}+3} \overline{(G_i \lfloor^{\mathsf{o}+4} r)},\ (\mu_q : \oplus^{\mathsf{o}+2}\{i : \overline{(G_i \lfloor^{\mathsf{o}+4} q)}\}_{i \in I})_{q \in \mathsf{deps}_s},\ (\mu_q : \oplus^{\mathsf{o}+3}\{i : \overline{(G_i \lfloor^{\mathsf{o}+4} q)}\}_{i \in I})_{q \in \mathsf{deps}_r},\ (\mu_q : \overline{G_i \lfloor^{\mathsf{o}+4} q})_{q \in \tilde{q} \setminus \mathsf{deps}_s \setminus \mathsf{deps}_r \setminus \{s,r\}}}\ (\oplus^\star)^*
}{\forall i \in I.\ 2_i \vdash \mu_s : \overline{(\!|S_i|\!)} \parr^{\mathsf{o}+1} \overline{(G_i \lfloor^{\mathsf{o}+4} s)},\ \mu_r : \oplus^{\mathsf{o}+2}\{i : (\!|S_i|\!) \otimes^{\mathsf{o}+3} \overline{(G_i \lfloor^{\mathsf{o}+4} r)}\}_{i \in I},\ (\mu_q : \oplus^{\mathsf{o}+2}\{i : \overline{(G_i \lfloor^{\mathsf{o}+4} q)}\}_{i \in I})_{q \in \mathsf{deps}_s},\ (\mu_q : \oplus^{\mathsf{o}+3}\{i : \overline{(G_i \lfloor^{\mathsf{o}+4} q)}\}_{i \in I})_{q \in \mathsf{deps}_r},\ (\mu_q : \overline{G_i \lfloor^{\mathsf{o}+4} q})_{q \in \tilde{q} \setminus \mathsf{deps}_s \setminus \mathsf{deps}_r \setminus \{s,r\}}}\ \oplus^\star
}{\begin{aligned}
&\mathsf{O}_{\tilde{q}}[G_s] = 1 \vdash \mu_s : \&^{\mathsf{o}}\{i : \overline{(\!|S_i|\!)} \parr^{\mathsf{o}+1} \overline{(G_i \lfloor^{\mathsf{o}+4} s)}\}_{i \in I}, && \text{(cf. (30))}\\
&\mu_r : \oplus^{\mathsf{o}+2}\{i : (\!|S_i|\!) \otimes^{\mathsf{o}+3} (G_i \lfloor^{\mathsf{o}+4} r)\}_{i \in I}, && \text{(cf. (31))}\\
&(\mu_q : \oplus^{\mathsf{o}+2}\{i : \overline{(G_i \lfloor^{\mathsf{o}+4} q)}\}_{i \in I})_{q \in \mathsf{deps}_s} && \text{(cf. (32))}\\
&(\mu_q : \oplus^{\mathsf{o}+3}\{i : \overline{(G_i \lfloor^{\mathsf{o}+4} q)}\}_{i \in I})_{q \in \mathsf{deps}_r} && \text{(cf. (33))}\\
&(\mu_q : \overline{G_{i'} \lfloor^{\mathsf{o}+4} q})_{q \in \tilde{q} \setminus \mathsf{deps}_s \setminus \mathsf{deps}_r \setminus \{s,r\}} && \text{(cf. (34))}
\end{aligned}}\ \&
$$

Fig. 12. Typing derivation used in the proof of Theorem 24.

We also need an assignment in the recursive context for every $X \in \widetilde{X_C}$, but not for Z.
Let $C' = C[\mu Z\ .\ []]$. Clearly, $G' \preceq_{C'} G$. Let us establish some facts about the recursion binders, priorities, and active participants related to C', G', and Z:

* $\widetilde{X_{C'}} = \mathsf{ctxbind}(C') = (\mathsf{ctxbind}(C), Z) = (\widetilde{X_C}, Z)$ (cf. Definition 30).
* $G_Z = \mathsf{recdef}(Z, G) = G'$, as proven by the context C' (cf. Definition 31).
* $\widetilde{Y_Z} = \mathsf{subbind}(\mu Z\ .\ G_Z, G) = \mathsf{ctxbind}(C) = \widetilde{X_C}$.
* $\mathsf{o}_{C'} = \mathsf{ctxpri}(C') = \mathsf{ctxpri}(C) = \mathsf{o}_C$, and $\mathsf{o}_Z = \mathsf{varpri}(Z, G) = \mathsf{ctxpri}^C(=)\mathsf{o}_C$, and hence $\mathsf{o}_{C'} = \mathsf{o}_Z$ (cf. Definition 32).
* $\tilde{q}_Z = \tilde{q}'$ (cf. Definition 33 and (35)).

Because $\widetilde{X_{C'}} = (\widetilde{X_C}, Z)$ and $\tilde{q}' = \tilde{q}_Z$, $\tilde{q}'$ is appropriate for the IH. We apply the IH on C', G', and $\tilde{q}'$ to obtain a typing for $\mathsf{O}_{\tilde{q}'}[G']$, where we immediately make use of the facts established above. We give the assignment to Z in the recursive context separate from those for the recursion variables in $\widetilde{X_C}$. Also, by Proposition 15, we can write the final unfolding on Z in the types separately.

$$
\begin{aligned}
\mathsf{O}_{\tilde{q}'}[G'] \vdash\ & \Big(X : \big(\mathsf{deepUnfold}(\overline{G_X \lfloor^{\mathsf{o}_X} q}, (Y, t_Y, \overline{G_Y \lfloor^{\mathsf{o}_Y} q})_{Y \in \widetilde{Y_X}})\big)_{q \in \tilde{q}_X}\Big)_{X \in \widetilde{X_C}},\\
& Z : \big(\mathsf{deepUnfold}(\overline{G' \lfloor^{\mathsf{o}_C} q}, (X, t_X, \overline{G_X \lfloor^{\mathsf{o}_X} q})_{X \in \widetilde{X_C}})\big)_{q \in \tilde{q}'};\\
& \Big(\mu_q : \mathsf{unfold}^{t_Z}(\mu Z\ .\ \mathsf{deepUnfold}(\overline{G' \lfloor^{\mathsf{o}_C} q}, (X, t_X, \overline{G_X \lfloor^{\mathsf{o}_X} q})_{X \in \widetilde{X_C}}))\Big)_{q \in \tilde{q}'}
\end{aligned}
$$

By assumption, we have

$$
t_Z = \max_{\mathsf{pr}} \Big(\mathsf{deepUnfold}(\overline{G' \lfloor^{\mathsf{o}_C} q}, (X, t_X, \overline{G_X \lfloor^{\mathsf{o}_X} q})_{X \in \widetilde{X_C}})\Big)_{q \in \tilde{q}'} + 1,
$$

so t_Z is clearly bigger than the maximum priority appearing in the types before unfolding. Hence, we can apply Rec to eliminate Z from the recursive context, and to fold the types, giving the typing of $\mathsf{O}_{\tilde{q}}[G_s] = \mu Z((\mu_q)_{q\in\tilde{q}'}) \,.\, \mathsf{O}_{\tilde{q}'}[G']$:

$$\mathsf{O}_{\tilde{q}}[G_s] \vdash \Big(X : \big(\mathsf{deepUnfold}(\overline{G_X \lfloor^{\mathsf{o}_X} q}, (Y, t_Y, \overline{G_Y \lfloor^{\mathsf{o}_Y} q})_{Y\in\widetilde{Y_X}})\big)_{q\in\tilde{q}_X}\Big)_{X\in\widetilde{X_C}};$$
$$\Big(\mu_q : \mu Z \,.\, \mathsf{deepUnfold}(\overline{G' \lfloor^{\mathsf{o}_C} q}, (X, t_X, \overline{G_X \lfloor^{\mathsf{o}_X} q})_{X\in\widetilde{X_C}})\Big)_{q\in\tilde{q}'}$$

In this typing, the type for μ_q for every $q \in \tilde{q}'$ concurs with (37). For every $q \in \tilde{q} \setminus \tilde{q}'$, we can add the type for μ_q in (38) by applying $\bullet$. This proves the thesis.

- *Recursive call*: $G_s = Z$ (line 10).
  Following similar reasoning as in the case of recursive call in the proof of Theorem 16, let us take stock of the types we expect for our orchestrator's channels.

  For each $q \in \tilde{q}$,

  for μ_q we expect
  $$\begin{aligned}
  &\mathsf{deepUnfold}(\overline{G_s \lfloor^{\mathsf{o}_C} q}, \ldots)\\
  &= \mathsf{deepUnfold}(\overline{Z \lfloor^{\mathsf{o}_C} q}, \ldots)\\
  &= \mathsf{deepUnfold}(\overline{Z}, \ldots)\\
  &= \mathsf{deepUnfold}(Z, (X, t_X, \overline{G_X \lfloor^{\mathsf{o}_X} q})_{X\in(\tilde{X}_1, Z, \widetilde{Y_Z})})\\
  &= \mathsf{deepUnfold}(Z, (X, t_X, \overline{G_X \lfloor^{\mathsf{o}_X} q})_{X\in(Z, \widetilde{Y_Z})})\\
  &= \mu Z \,.\, (\uparrow^{t_Z} \mathsf{deepUnfold}(\overline{G_Z \lfloor^{\mathsf{o}_Z} q}, (X, t_X, \overline{G_X \lfloor^{\mathsf{o}_X} q})_{X\in\widetilde{Y_Z}}))
  \end{aligned} \tag{39}$$

  Also, we need an assignment in the recursive context for every $X \in \widetilde{X_C}$. By Lemma 13, $\tilde{q} = \tilde{q}_Z$. Hence, for Z, the assignment should be as follows:

  $$Z : \Big(\mathsf{deepUnfold}(\overline{G_Z \lfloor^{\mathsf{o}_Z} q}, (X, t_X, \overline{G_X \lfloor^{\mathsf{o}_X} q})_{X\in\widetilde{Y_Z}})\Big)_{q\in\tilde{q}} \tag{40}$$

  We apply Var to obtain the typing of $\mathsf{O}_{\tilde{q}}[G_s]$, where we make us the rule's allowance for an arbitrary recursive context up to the assignment to Z. Var is applicable, because the types are recursive definitions on Z, concurring with the types assigned to Z, and lifted by a common lifter t_Z.

  $$\frac{}{\begin{aligned}
  \mathsf{O}_{\tilde{q}}[G_s] = X\langle(\mu_q)_{q\in\tilde{q}}\rangle \vdash &\Big(X : \big(\mathsf{deepUnfold}(\overline{G_X \lfloor^{\mathsf{o}_X} q}, (Y, t_Y, \overline{G_Y \lfloor^{\mathsf{o}_Y} q})_{Y\in\widetilde{Y_X}})\big)_{q\in\tilde{q}_X}\Big)_{X\in\widetilde{X_C}\setminus(Z)},\\
  &Z : \Big(\mathsf{deepUnfold}(\overline{G_Z \lfloor^{\mathsf{o}_Z} q}, (X, t_X, \overline{G_X \lfloor^{\mathsf{o}_X} q})_{X\in\widetilde{Y_Z}})\Big)_{q\in\tilde{q}};\\
  &\big(\mu_q : \mu Z \,.\, (\uparrow^{t_Z} \mathsf{deepUnfold}(\overline{G_Z \lfloor^{\mathsf{o}_Z} q}, (X, t_X, \overline{G_X \lfloor^{\mathsf{o}_X} q})_{X\in\widetilde{Y_Z}}))\big)_{q\in\tilde{q}}
  \end{aligned}}\ \textsc{Var}$$

  In this typing, the types of μ_q for each $q \in \tilde{q}$ concur with the expected types in (39), and the assignment to Z in the recursive context concurs with (40). This proves the thesis. □

4.4.2. Orchestrators and centralized compositions of routers are behaviorally equivalent

First, we formalize what we mean with a centralized composition of routers, which we call a *hub of routers*. A hub of routers is just a specific composition of routers, formalized as the centralized composition of the routers of all a global type's participants synthesized from the global type:

Definition 37 *(Hub of a global type).* Given global type G, we define the *hub of routers* of G as follows:

$$\mathcal{H}_G := (\boldsymbol{\nu}\, p_q q_p)_{p,q\in\mathsf{prt}(G)}\big(\textstyle\prod_{p\in\mathsf{prt}(G)} \mathcal{R}_p\big)$$

Hubs of routers can be typed using local projection (cf. Definition 23), identical to the typing of orchestrators (cf. Theorem 24):

Theorem 25. *For relative well-formed global type G and priority o,*

$$\mathcal{H}_G \vdash \emptyset;\ (\mu_p : \overline{(G \lfloor^{\mathsf{o}} p)})_{p\in\mathsf{prt}(G)}.$$

Proof. By the typability of routers (Theorem 11) and the duality of the types of the endpoints connecting pairs of routers (Theorem 9). □

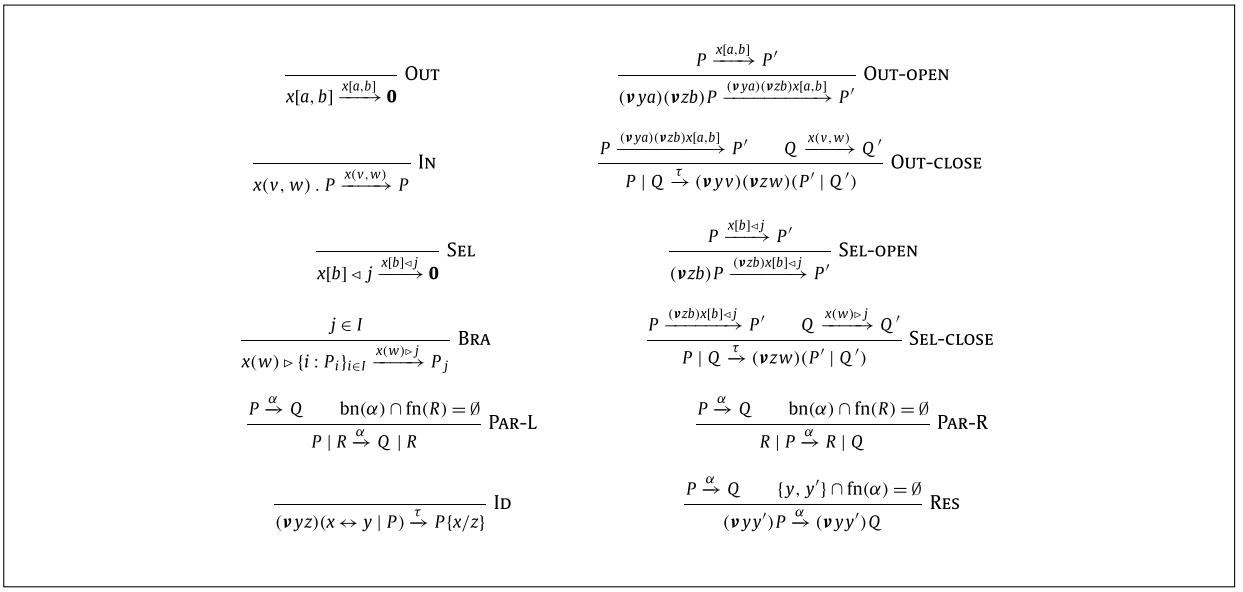

Fig. 13. Labeled transition system for APCP (cf. Definition 38).

To state the behavioral equivalence of orchestrators and hubs of routers, we first define the specific behavioral equivalence we desire. To this end, we first define a labeled transition system (LTS) for APCP:

Definition 38 *(LTS for APCP).* We define the labels α for transitions for processes as follows:

$\alpha ::= \tau$	communication		
$\mid x[a,b]$	output	$\mid (\boldsymbol{\nu} ya)(\boldsymbol{\nu} zb)x[a,b]$	bound output
$\mid x[b] \lhd j$	selection	$\mid (\boldsymbol{\nu} zb)x[b] \lhd j$	bound selection
$\mid x(v,w)$	input	$\mid x(w) \rhd j$	branch

The relation *labeled transition* ($P \xrightarrow{\alpha} Q$) is then defined by the rules in Fig. 13.

Proposition 26. *$P \longrightarrow Q$ if and only if $P \xrightarrow{\tau} Q$.*

As customary, we write $\Rightarrow$ for the reflexive, transitive closure of $\xrightarrow{\tau}$, and we write $\stackrel{\alpha}{\Rightarrow}$ for $\Rightarrow\xrightarrow{\alpha}\Rightarrow$ if $\alpha \neq \tau$ and for $\Rightarrow$ otherwise.

We can now define the behavioral equivalence we desire:

Definition 39 *(Weak bisimilarity).* A binary relation $\mathbb{B}$ on processes is a *weak bisimulation* if whenever $(P, Q) \in \mathbb{B}$,

- $P \xrightarrow{\alpha} P'$ implies that there is Q' such that $Q \stackrel{\alpha}{\Rightarrow} Q'$ and $(P', Q') \in \mathbb{B}$, and
- $Q \xrightarrow{\alpha} Q'$ implies that there is P' such that $P \stackrel{\alpha}{\Rightarrow} P'$ and $(P', Q') \in \mathbb{B}$.

Two processes P and Q are *weakly bisimilar* if there exists a weak bisimulation $\mathbb{B}$ such that $(P, Q) \in \mathbb{B}$.

Our equivalence result shall relate an orchestrator and a hub on a single but arbitrary channel. In order to isolate such a channel, we place the orchestrator and hub of routers in an evaluation context consisting of restrictions and parallel compositions with arbitrary processes, such that it connects all but one of the orchestrator's or hub's channels. For example, given a global type G and implementations $P_q \vdash \emptyset; \mu_q : G \downarrow^0 q$ for every participant $q \in \mathsf{prt}(G) \setminus \{p\}$, we could use the following evaluation context:

$$E := (\boldsymbol{\nu} \mu_q q_\mu)_{q \in \mathsf{prt}(G)\setminus\{p\}} \big(\textstyle\prod_{q \in \mathsf{prt}(G)\setminus\{p\}} P_q \mid [\,] \big)$$

Replacing the hole in this evaluation context with the orchestrator or hub of routers of G leaves one channel free: the channel μ_p for the implementation of p. Now, we can observe the behavior of these two processes on μ_p.

In what follows we write $\mathcal{O}_G^{\tilde{q}}$ instead of $\mathsf{O}_{\tilde{q}}[G]$. When we appeal to router and orchestrator synthesis, we often omit the parameter $\tilde{q}$. That is, we write $[\![G]\!]_p$ instead of $[\![G]\!]_p^{\tilde{q}}$, and $\mathcal{O}_G$ instead of $\mathcal{O}_G^{\tilde{q}}$.

Theorem 27. *Suppose given a relative well-formed global type G. Let $\mathcal{H}_G$ be the hub of routers of G (Definition 37) and take the orchestrator $\mathcal{O}_G^{\mathsf{prt}(G)}$ of G (Definition 36). Let $p \in \mathsf{prt}(G)$, and let E be an evaluation context such that $E[\mathcal{H}_G] \vdash \emptyset; \mu_p : (\overline{G \upharpoonright^{\circ} p})$. Then, $E[\mathcal{H}_G]$ and $E[\mathcal{O}_G]$ are weakly bisimilar (Definition 39).*

We first give an intuition for the proof of Theorem 27 and its ingredients, after which we give the proof and detail the ingredients. The proof is by coinduction, i.e., by exhibiting a weak bisimulation $\mathbb{B}$ that contains the pair $(E[\mathcal{H}_G], E[\mathcal{O}_G])$. To construct $\mathbb{B}$ and prove that it is a weak bisimulation we require the following:

- We define a function that, given a global type G and a *starting relation* $\mathbb{B}_0$, computes a corresponding *candidate relation*. This function is denoted $\mathbb{B}(G, \mathbb{B}_0)$ (Definition 40).
- Suppose $G \xrightarrow{\beta_1} \dots \xrightarrow{\beta_k} G'$, with $k \geq 0$. Given some starting relation $\mathbb{B}_0$, we want to show that the relation obtained from $\mathbb{B}(G', \mathbb{B}_0)$ is a weak bisimulation, for which we need to assert that $\mathbb{B}_0$ is an appropriate starting relation. To this end, we define a function that computes a *consistent* starting relation for a bisimulation relation, given a pair (P, Q) of processes and a participant p of G. This function is denoted $\langle G \xrightarrow{\beta_1} \dots \xrightarrow{\beta_k} G', (P, Q), p\rangle$ (Definition 41).
- The property that processes in such a consistent starting relation follow a pattern of specific labeled transitions, passing through a context containing the router of p or the orchestrator (Lemma 28).
- The property that the relation obtained from $\mathbb{B}(G', \mathbb{B}_0, p)$ is a weak bisimulation, given the consistent starting relation $\mathbb{B}_0 = \langle G \xrightarrow{\beta_1} \dots \xrightarrow{\beta_k} G', (E[\mathcal{H}_G], E[\mathcal{O}_G]), p\rangle$ (Lemma 29).

Theorem 27 follows from these definitions and results:

Proof of Theorem 27. Let $\mathbb{B} = \mathbb{B}(G, \mathbb{B}_0)$, where $\mathbb{B}_0 = \langle G, (E[\mathcal{H}_G], E[\mathcal{O}_G]), p\rangle$. By Lemma 29, $\mathbb{B}$ is a weak bisimulation. Because $(E[\mathcal{H}_G], E[\mathcal{O}_G]) \in \mathbb{B}_0 \subseteq \mathbb{B}$, it then follows that $E[\mathcal{H}_G]$ and $E[\mathcal{O}_G]$ are weakly bisimilar. □

We setup some notations:

Notation 4. We adopt the following notational conventions.

- We write Proc to denote the set of all typable APCP processes.
- In the LTS for APCP (Definition 38), we simplify labels: we write an overlined variant for output and selection (e.g., for $(\boldsymbol{\nu} ab)\mu_p[a] \triangleleft \ell$ we write $\overline{\mu_p} \triangleleft \ell$), and omit continuation channels for input and branching (e.g., for $\mu_p(a) \triangleright \ell$ we write $\mu_p \triangleright \ell$).
- Also, we write $P \xRightarrow{\alpha_1 \dots \alpha_n} Q$ rather than $P \xRightarrow{\alpha_1} P_1 \xRightarrow{\alpha_2} P_2 \dots \xRightarrow{\alpha_n} Q$.
- We write $\tilde{\alpha}$ to denote a sequence of labels, e.g., if $\tilde{\alpha} = \alpha_1 \dots \alpha_n$ then $\xRightarrow{\tilde{\alpha}} = \xRightarrow{\alpha_1 \dots \alpha_n}$. If $\tilde{\alpha} = \epsilon$ (empty sequence), then $\xRightarrow{\tilde{\alpha}} = \Rightarrow$.

The following function defines a relation on processes, which we will use as the weak bisimulation between $E[\mathcal{H}_G]$ and $E[\mathcal{O}_G]$:

Definition 40 *(Candidate relation).* Let G be a global type and let p be a participant of G. Also, let $\mathbb{B}_0 \subseteq \mathsf{Proc} \times \mathsf{Proc}$ denote a relation on processes. We define a *candidate relation* for a weak bisimulation of the hub and orchestrator of G observed on μ_p starting at $\mathbb{B}_0$, by abuse of notation denoted $\mathbb{B}(G, \mathbb{B}_0, p)$. The definition is inductive on the structure of G:

- $G = \bullet$. Then $\mathbb{B}(G, \mathbb{B}_0, p) = \mathbb{B}_0$.
- $G = s \twoheadrightarrow r\{i\langle S_i\rangle . G_i\}_{i\in I}$. We distinguish four cases, depending on the involvement of p:
  - $p = s$. For every $i \in I$, let

$$\mathbb{B}_1^i = \{(P_1, Q_1) \mid \exists (P_0, Q_0) \in \mathbb{B}_0 \text{ s.t. } P_0 \xrightarrow{\mu_p \triangleright i} \Rightarrow P_1 \text{ and } Q_0 \xrightarrow{\mu_p \triangleright i} \Rightarrow Q_1\};$$

$$\mathbb{B}_2^i = \{(P_2, Q_2) \mid \exists (P_1, Q_1) \in \mathbb{B}_1^i \text{ s.t. } P_1 \xrightarrow{\mu_p(y)} \Rightarrow P_2 \text{ and } Q_1 \xrightarrow{\mu_p(y)} \Rightarrow Q_2\}$$

  Then

$$\mathbb{B}(G, \mathbb{B}_0, p) = \mathbb{B}_0 \cup \textstyle\bigcup_{i\in I}(\mathbb{B}_1^i \cup \mathbb{B}(G_i, \mathbb{B}_2^i, p)).$$

– $p = r$. For every $i \in I$, let

$$\mathbb{B}_1^i = \{(P_1, Q_1) \mid \exists (P_0, Q_0) \in \mathbb{B}_0 \text{ s.t. } P_0 \xrightarrow{\overline{\mu_p} \triangleleft i} \Rightarrow P_1 \text{ and } Q_0 \xrightarrow{\overline{\mu_p} \triangleleft i} \Rightarrow Q_1\};$$

$$\mathbb{B}_2^i = \{(P_2, Q_2) \mid \exists (P_1, Q_1) \in \mathbb{B}_1 \text{ and } y \text{ s.t. } P_1 \xrightarrow{\overline{\mu_p}[y]} \Rightarrow P_2 \text{ and } Q_1 \xrightarrow{\overline{\mu_p}[y]} \Rightarrow Q_2\}.$$

Then

$$\mathbb{B}(G, \mathbb{B}_0, p) = \mathbb{B}_0 \cup \textstyle\bigcup_{i \in I} (\mathbb{B}_1^i \cup \mathbb{B}(G_i, \mathbb{B}_2^i, p)).$$

– $p \notin \{s, r\}$ and $\mathrm{hdep}(p, s, G)$ or $\mathrm{hdep}(p, r, G)$. For every $i \in I$, let

$$\mathbb{B}_1^i = \{(P_1, Q_1) \mid \exists (P_0, Q_0) \in \mathbb{B}_0 \text{ s.t. } P_0 \xrightarrow{\overline{\mu_p} \triangleleft i} \Rightarrow P_1 \text{ and } Q_0 \xrightarrow{\overline{\mu_p} \triangleleft i} \Rightarrow Q_1\}$$

Then

$$\mathbb{B}(G, \mathbb{B}_0, p) = \mathbb{B}_0 \cup \textstyle\bigcup_{i \in I} \mathbb{B}(G_i, \mathbb{B}_1^i, p).$$

– $p \notin \{s, r\}$ and neither $\mathrm{hdep}(p, s, G)$ nor $\mathrm{hdep}(p, r, G)$. Then

$$\mathbb{B}(G, \mathbb{B}_0, p) = \mathbb{B}(G_j, \mathbb{B}_0, p)$$

for any $j \in I$.

- $G = \mu X \,.\, G'$. Then $\mathbb{B}(G, \mathbb{B}_0, p) = \mathbb{B}(G'\{\mu X \,.\, G'/X\}, \mathbb{B}_0, p)$.
- $G = \mathsf{skip} \,.\, G'$. Then $\mathbb{B}(G, \mathbb{B}_0, p) = \mathbb{B}(G', \mathbb{B}_0, p)$.

The function $\mathbb{B}(G, \mathbb{B}_0, p)$ constructs a relation between processes by following labeled transitions on μ_p that concur with the expected behavior of p's router and the orchestrator depending on the shape of G. For example, for $G = s \twoheadrightarrow p\{i\langle S_i\rangle \,.\, G_i\}_{i \in I}$, for each $i \in I$, the function constructs $\mathbb{B}_1^i$ containing the processes reachable from $\mathbb{B}_0$ through a transition labeled $\overline{\mu_p} \triangleleft i$ (selection of the label chosen by s), and $\mathbb{B}_2^i$ containing the processes reachable from $\mathbb{B}_0$ through a transition labeled $\overline{\mu_p}[y]$ (output of the endpoint sent by s); the resulting relation then consists of $\mathbb{B}_0$ and, for each $i \in I$, $\mathbb{B}_1^i$ and $\mathbb{B}(G_i, \mathbb{B}_2^i, p)$ (i.e., the candidate relation for G_i starting with B_2^i). Since we are interested in a *weak* bisimulation, the τ-transitions of one process do not need to be simulated by related processes. Hence, e.g., if $(P, Q) \in \mathbb{B}_0$ and $P \xrightarrow{\tau} P'$ and $Q \xrightarrow{\tau} Q'$, then $\{(P, Q), (P', Q), (P, Q'), (P', Q')\} \subseteq \mathbb{B}(G, \mathbb{B}_0, p)$. This way, we only *synchronize* related processes when they can both take the same labeled transition.

We intend to show that, if $G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_k} G'$, the function $\mathbb{B}(G', \mathbb{B}_0, p)$ constructs a weak bisimulation. However, for this to hold, the starting relation $\mathbb{B}_0$ cannot be arbitrary: the pairs of processes in $\mathbb{B}_0$ have to be reachable from $E[\mathcal{H}_G]$ and $E[\mathcal{O}_G]$ through labeled transitions that concur with the transitions from G to G'. Moreover, the processes must have "passed through" evaluation contexts containing the router for p at G' and the orchestrator at G'. The following defines a *consistent starting relation*, parametric on k, that satisfies these requirements. Note that for constructing the relation $\mathbb{B}$, we only need the following definition for $k = 0$. However, in the proof that $\mathbb{B}$ is a weak bisimulation we need to generalize it to $k \geq 0$ to assure that the starting relation of coinductive steps is consistent.

Definition 41 *(Consistent starting relation).* Let $G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_k} G'$ (with $k \geq 0$) be a sequence of labeled transitions from G to G' including the intermediate global types (cf. Definition 35) and let p be a participant of G. Also, let (P, Q) be a pair of initial processes. We define the *consistent starting relation* for observing the hub and orchestrator of G' on μ_p starting at (P, Q) after the transitions from G to G', denoted $\langle G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_k} G', (P, Q), p\rangle$. The definition is inductive on the number k of transitions:

- $k = 0$. Then $\langle G, (P, Q), p\rangle = \{(P', Q') \mid P \Rightarrow P' \text{ and } Q \Rightarrow Q'\}$.
- $k = k' + 1$. Then

$$\langle G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_{k'}} G_{k'} \xrightarrow{\beta_k} G_k, (P, Q), p\rangle =$$

$$\{(P_k, Q_k) \mid \exists (P_{k'}, Q_{k'}) \in \langle G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_{k'}} G_{k'}, (P, Q), p\rangle$$
$$\text{s.t. } (\quad (\exists C \text{ s.t. } P_{k'} \overset{\tilde{\alpha}}{\Rightarrow} C[[\![G_k]\!]_p] \Rightarrow P_k)$$
$$\text{and } (\exists D \text{ s.t. } Q_{k'} \overset{\tilde{\alpha}}{\Rightarrow} D[\mathcal{O}_{G_k}] \Rightarrow Q_k))\},$$

where $\tilde{\alpha}$ depends on $\beta_k = s\rangle r : j\langle S_j\rangle$ and $G_{k'}$ (in unfolded form if $G_{k'} = \mu X \,.\, G'_{k'}$):

– If $p = s$, then $\tilde{\alpha} = \mu_p \triangleright j\, \mu_p(y)$.

- If $p = r$, then $\tilde{\alpha} = \overline{\mu_p} \triangleleft j\, \overline{\mu_p}[y]$.
- If $p \notin \{s, r\}$ and $\mathrm{hdep}(p, s, G_k)$ or $\mathrm{hdep}(p, r, G_k)$, then $\tilde{\alpha} = \overline{\mu_p} \triangleleft j$.
- If $p \notin \{s, r\}$ and neither $\mathrm{hdep}(p, s, G_k)$ nor $\mathrm{hdep}(p, r, G_k)$, then $\tilde{\alpha} = \epsilon$.

Lemma 28. *Let G be a relative well-formed global type such that $G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_k} G'$ for $k \geq 0$ and let p be a participant of G. Also, let E be an evaluation context such that $\mathrm{fn}(E) = \{\mu_p\}$. Then there exists $\tilde{\alpha}$ such that, for every $(P, Q) \in \langle G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_k} G', (E[\mathcal{H}_G], E[\mathcal{O}_G]), p\rangle$,*

- $E[\mathcal{H}_G] \overset{\tilde{\alpha}}{\Rightarrow} C\big[[\![G']\!]_p\big] \Rightarrow P$ *where C is an evaluation context without an output or selection on μ_p; and*
- $E[\mathcal{O}_G] \overset{\tilde{\alpha}}{\Rightarrow} D\big[\mathcal{O}_{G'}\big] \Rightarrow Q$ *where D is an evaluation context without an output or selection on μ_p.*

Proof. By induction on k. In the base case ($k = 0$), we have $G = G'$, so $E[\mathcal{H}_G] = C[[\![G']\!]_p] \Rightarrow P$ and $E[\mathcal{O}] = D[\mathcal{O}_{G'}] \Rightarrow Q$.

For the inductive case ($k = k' + 1$), we detail the representative case where

$$G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_{k'}} G_{k'} = p \twoheadrightarrow s\{i\langle S_i\rangle \,.\, G'_i\}_{i\in I} \xrightarrow{p\rangle s : i'\langle S_{i'}\rangle} G'$$

for some $i' \in I$. By the IH, for every $(P_{k'}, Q_{k'}) \in \langle G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_{k'}} G_{k'}, (E[\mathcal{H}_G], E[\mathcal{O}_G]), p\rangle$, there exists $\tilde{\alpha}'$ such that $E[\mathcal{H}_G] \overset{\tilde{\alpha}'}{\Rightarrow} C'[[\![G_{k'}]\!]^p] \Rightarrow P_{k'}$ and $E[\mathcal{O}_G] \overset{\tilde{\alpha}'}{\Rightarrow} D'[\mathcal{O}_{G_{k'}}] \Rightarrow Q_{k'}$ where C' and D' are without output or selection on μ_p. Take any $(P, Q) \in \langle G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_{k'}} G_{k'} \xrightarrow{s\rangle p : i'\langle S_{i'}\rangle} G', (E[\mathcal{H}_G], E[\mathcal{O}_G]), p\rangle$. By definition, there exists $(P_{k'}, Q_{k'}) \in \langle G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_{k'}} G_{k'}, (E[\mathcal{H}_G], E[\mathcal{O}_G]), p\rangle$ such that

$$P_{k'} \xrightarrow{\overline{\mu_p} \triangleleft i' \overline{\mu_p}[y]} C[[\![G']\!]^p] \Rightarrow P \text{ and } Q_{k'} \xrightarrow{\overline{\mu_p} \triangleleft i' \overline{\mu_p}[y]} D[\mathcal{O}_{G'}] \Rightarrow Q$$

where there are no outputs or selection on μ_p in C and D. Let $\tilde{\alpha} = \tilde{\alpha}'\, \overline{\mu_p \triangleleft i'}\, \overline{\mu_p}[y]$. Then $E[\mathcal{H}_G] \overset{\tilde{\alpha}}{\Rightarrow} C[[\![G']\!]_p] \Rightarrow P$ and $E[\mathcal{O}_G] \overset{\tilde{\alpha}}{\Rightarrow} D[\mathcal{O}_{G'}] \Rightarrow Q$. □

Lemma 29. *Let G be a relative well-formed global type such that $G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_k} G'$ (with $k \geq 0$) and let p be a participant of G. Also, let E be an evaluation context such that $\mathrm{fn}(E) = \{\mu_p\}$. Then the relation $\mathbb{B}(G', \mathbb{B}_0)$, with $\mathbb{B}_0 = \langle G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_k} G', (E[\mathcal{H}_G], E[\mathcal{O}_G]), p\rangle$, is a weak bisimulation (cf. Definition 39).*

Proof. By coinduction on the structure of G'; there are four cases (communication, recursion, skip, and $\bullet$). We only detail the interesting case of communication, which is the only case which involves transitions with labels other than τ. There are four subcases depending on the involvement of p in the communication (p is sender, p is recipient, p depends on the communication, or p does not depend on the communication). In each subcase, the proof follows the same pattern, so as a representative case, we detail when p is the recipient of the communication, i.e., $G' = s \twoheadrightarrow p\{i\langle S_i\rangle \,.\, G'_i\}_{i\in I}$. Recall

$$[\![G']\!]_p = p_s \triangleright \Big\{i{:}\, \overline{\mu_p} \triangleleft i \cdot (\overline{p_q} \triangleleft i)_{q\in \mathsf{deps}} \cdot p_s(v) \,.\, \overline{\mu_p}[w] \cdot (v \leftrightarrow w \mid [\![G'_i]\!]_p)\Big\}_{i\in I}, \qquad \text{(Algorithm 1 line 6)}$$

$$[\![G']\!]_s = \mu_p \triangleright \Big\{i{:}\, \overline{s_p} \triangleleft i \cdot (\overline{s_q} \triangleleft i)_{q\in \mathsf{deps}} \cdot \mu_s(v) \,.\, \overline{s_p}[w] \cdot (v \leftrightarrow w \mid [\![G'_i]\!]_s)\Big\}_{i\in I}, \qquad \text{(Algorithm 1 line 5)}$$

$$\mathcal{O}_{G'} = \mu_s \triangleright \{i{:}\, \overline{\mu_p} \triangleleft i \cdot (\overline{\mu_q} \triangleleft i)_{q\in \mathsf{deps}} \cdot \mu_s(v) \,.\, \overline{\mu_p}[w] \cdot (v \leftrightarrow w \mid \mathcal{O}_{G'_i})\}_{i\in I}. \qquad \text{(Algorithm 2 line 3)}$$

Let $\mathbb{B} = \mathbb{B}(G', \mathbb{B}_0)$. We have $\mathbb{B} = \mathbb{B}_0 \cup \bigcup_{i\in I}(\mathbb{B}^i_1 \cup \mathbb{B}(G'_i, \mathbb{B}^i_2))$ with $\mathbb{B}^i_1$ and $\mathbb{B}^i_2$ as defined above. Take any $(P, Q) \in \mathbb{B}$; we distinguish cases depending on the subset of $\mathbb{B}$ to which (P, Q) belongs:

- $(P, Q) \in \mathbb{B}_0$. By Lemma 28, we have $E[\mathcal{H}_G] \overset{\tilde{\alpha}}{\Rightarrow} C[[\![G']\!]_p] \Rightarrow P$ and $E[\mathcal{O}] \overset{\tilde{\alpha}}{\Rightarrow} D[\mathcal{O}_{G'}] \Rightarrow Q$, where C and D do not contain an output or selection on μ_p.

  Suppose $P \xrightarrow{\alpha} P'$; we need to exhibit a matching weak transition from Q. By assumption, there are no outputs or selections on μ_p in C and D. Since there are no outputs or selections on μ_p in C, by definition of $[\![G']\!]_p$, we need only consider two cases for α:
  - $\alpha = \tau$. We have $Q \Rightarrow Q$, so $Q \overset{\tau}{\Rightarrow} Q$. Since $C[[\![G']\!]_p] \Rightarrow P'$ and $D[\mathcal{O}_{G'}] \Rightarrow Q$, we have $(P', Q) \in \mathbb{B}_0 \subseteq \mathbb{B}$.
  - $\alpha = \overline{\mu_p} \triangleleft j$ for some $j \in I$. To enable this transition, which originates from p's router, somewhere in the τ-transitions between $C[[\![G']\!]_p]$ and P the label j was received on p_s, sent by the router of s on s_p. For this to happen, the label j was received on μ_s, sent from the context on s_μ. Since $\mathcal{H}_G$ and $\mathcal{O}$ are embedded in the same context, the communication of j between s_μ and μ_s can also take place after a number of τ-transitions from $D[\mathcal{O}_{G'}]$, after which the selection of j on μ_p becomes enabled. Hence, since there are no outputs or selection on μ_p in D, we

have $Q \Rightarrow Q_0 \xrightarrow{\overline{\mu_p} \triangleleft j} Q'$. We have $D[\mathcal{O}_{G'}] \Rightarrow Q_0$, so $(P, Q_0) \in \mathbb{B}_0$. Since $P \xrightarrow{\overline{\mu_p} \triangleleft j} \Rightarrow P'$ and $Q_0 \xrightarrow{\overline{\mu_p} \triangleleft j} \Rightarrow Q'$, we have $(P', Q') \in \mathbb{B}_1^j \subseteq \mathbb{B}'$.

Now suppose $Q \xrightarrow{\alpha} Q'$; we need to exhibit a matching weak transition from P. Again, we need only consider two cases for α:

- $\alpha = \tau$. Analogous to the similar case above.
- $\alpha = \overline{\mu_p} \triangleleft j$ for some $j \in I$. To enable this transition, which originates from the orchestrator, somewhere in the τ-transitions between $D[\mathcal{O}_{G'}]$ and Q the label j was received on μ_s, sent from the context on s_μ. Hence, this communication can also take place after a number of transitions from $E[\mathcal{H}_G]$, where the label is received by the router of s. After this, from $C[[\![G']\!]_p]$, the router of s forwards j to p's router (communication between s_p and p_s), enabling the selection of j on μ_p in p's router. Hence, since there are no outputs or selections in C, we have $P \Rightarrow P_0 \xrightarrow{\overline{\mu_p} \triangleleft j} P'$. We have $C[[\![G']\!]_p] \Rightarrow P_0$, so $(P_0, Q) \in \mathbb{B}_0$. Since $P_0 \xrightarrow{\overline{\mu_p} \triangleleft j} \Rightarrow P'$ and $Q \xrightarrow{\overline{\mu_p} \triangleleft j} \Rightarrow Q'$, we have $(P', Q') \in \mathbb{B}_1^j \subseteq \mathbb{B}$.

- $(P, Q) \in \mathbb{B}_1^j$ for some $j \in I$. We have $E[\mathcal{H}_G] \overset{\tilde{\alpha}}{\Rightarrow} C[[\![G']\!]_p] \Rightarrow P_0 \xrightarrow{\overline{\mu_p} \triangleleft j} \Rightarrow P$ and $E[\mathcal{O}_G] \overset{\tilde{\alpha}}{\Rightarrow} D[\mathcal{O}_{G'}] \Rightarrow Q_0 \xrightarrow{\overline{\mu_p} \triangleleft j} \Rightarrow Q$ where $(P_0, Q_0) \in \mathbb{B}_0$. Since we have already observed the selection of j on μ_p from both the hub and the orchestrator, we know that the routers of p and s are in branch j, and similarly the orchestrator is in branch j.
  Suppose $P \xrightarrow{\alpha} P'$. To exhibit a matching weak transition from Q we only need to consider two cases for α:
  - $\alpha = \tau$. We have $Q \overset{\tau}{\Rightarrow} Q$, and $P_0 \xrightarrow{\overline{\mu_p} \triangleleft j} \Rightarrow P'$ and $Q_0 \xrightarrow{\overline{\mu_p} \triangleleft j} \Rightarrow Q$, so $(P', Q) \in \mathbb{B}_1^j \subseteq \mathbb{B}$.
  - $\alpha = \overline{\mu_p}[y]$ for some y. The observed output of some y on μ_p must originate from p's router. This output is only enabled after receiving some v over p_s, which must be sent by the router of s over s_p. The output by the router of s is only enabled after receiving some v over μ_s, sent by the context over s_μ. Since the hub and the orchestrator are embedded in the same context, the communication of v from s_μ to μ_s can also occur (or has already occurred) for the orchestrator. After this, the output of y over μ_p is enabled in the orchestrator, i.e., $Q \Rightarrow Q_1 \xrightarrow{\overline{\mu_p}[y]} Q'$. We have $Q_0 \xrightarrow{\overline{\mu_p} \triangleleft j} \Rightarrow Q_1$, so $(P, Q_1) \in \mathbb{B}_1^j$. Since $P \xrightarrow{\overline{\mu_p}[y]} \Rightarrow P'$ and $Q_1 \xrightarrow{\overline{\mu_p}[y]} \Rightarrow Q'$, we have $(P', Q') \in \mathbb{B}_2^j$. By definition, $\mathbb{B}_2^j \subseteq \mathbb{B}(G'_j, B_2^j) \subseteq \mathbb{B}$, so $(P', Q') \in \mathbb{B}$.

  Now suppose $Q \xrightarrow{\alpha} Q'$. To exhibit a matching weak transition from P we only need to consider two cases for α:
  - $\alpha = \tau$. Analogous to the similar case above.
  - $\alpha = \overline{\mu_p}[y]$ for some y. The observed output of some y on μ_p must originate from the orchestrator. This output is only enabled after receiving some v over μ_s, sent by the context of s_μ. Since the hub and the orchestrator are embedded in the same context, the communication of v from s_μ to μ_s can also occur (or has already occurred) for the router of s. After this, the router of s sends another channel v' over s_p, received by p's router on p_s. This enables the output of y on μ_p by p's router, i.e., $P \Rightarrow P_1 \xrightarrow{\overline{\mu_p}[y]} P'$. We have $P_0 \xrightarrow{\overline{\mu_p} \triangleleft j} \Rightarrow P_1$, so $(P_1, Q) \in \mathbb{B}_1^j$. Since $P \xrightarrow{\overline{\mu_p}[y]} \Rightarrow P'$ and $Q_1 \xrightarrow{\overline{\mu_p}[y]} \Rightarrow Q'$, we have $(P', Q') \in \mathbb{B}_2^j$. As above, this implies that $(P', Q') \in \mathbb{B}$.
- For some $j \in I$, $(P, Q) \in \mathbb{B}(G'_j, \mathbb{B}_2^j)$. The thesis follows from proving that $\mathbb{B}(G'_j, \mathbb{B}_2^j)$ is a weak bisimulation. For this, we want to appeal to the coinduction hypothesis, so we have to show that $\mathbb{B}_2^j = \langle G \xrightarrow{\beta_1} \dots \xrightarrow{\beta_k} G' \xrightarrow{s \rangle p:j \langle S_j \rangle} G'_j, (E[\mathcal{H}_G], E[\mathcal{O}]), p \rangle$. We prove that $(P_2, Q_2) \in \mathbb{B}_2^j$ if and only if $(P_2, Q_2) \in \langle G \xrightarrow{\beta_1} \dots \xrightarrow{\beta_k} G' \xrightarrow{s \rangle p:j \langle S_j \rangle} G'_j, (E[\mathcal{H}_G], E[\mathcal{O}]), p \rangle$, i.e., we prove both directions of the bi-implication:
  - Take any $(P_2, Q_2) \in \mathbb{B}_2^j$. We have $E[\mathcal{H}_G] \overset{\tilde{\alpha}}{\Rightarrow} C[[\![G']\!]_p] \Rightarrow P_0 \xrightarrow{\overline{\mu_p} \triangleleft j} \Rightarrow P_1 \xrightarrow{\overline{\mu_p}[y]} \Rightarrow P_2$ and $E[\mathcal{O}] \overset{\tilde{\alpha}}{\Rightarrow} D[\mathcal{O}_{G'}] \Rightarrow Q_0 \xrightarrow{\overline{\mu_p} \triangleleft j} \Rightarrow Q_1 \xrightarrow{\overline{\mu_p}[y]} \Rightarrow Q_2$, where $(P_0, Q_0) \in \mathbb{B}_0$ and $(P_1, P_1) \in \mathbb{B}_1^j$.
    By definition, somewhere during the transitions from $C[[\![G']\!]_p]$ to P_1, we find $C'[[\![G'_j]\!]_p]$, which may then further reduce by τ-transitions towards P_2. As soon as we do find $C'[[\![G'_j]\!]_p]$, the output on μ_p is available, and the selection on μ_p has already occurred or is still available. Because they are asynchronous actions, we can observe the selection and output on μ_p as soon as they are available, before further reducing p's router. Hence, we can observe $C[[\![G']\!]_p] \Rightarrow \xrightarrow{\overline{\mu_p} \triangleleft j} \Rightarrow \xrightarrow{\overline{\mu_p}[y]} \Rightarrow C''[[\![G'_j]\!]_p] \Rightarrow P_2$, i.e.,

    $$E[\mathcal{H}_G] \overset{\tilde{\alpha}}{\Rightarrow} C[[\![G']\!]_p] \xrightarrow{\overline{\mu_p} \triangleleft j \, \overline{\mu_p}[y]} C''[[\![G'_j]\!]_p] \Rightarrow P_2.$$

    By definition, $[\![G'_j]\!]_p$ has no output or selection on μ_p available, so there are no outputs or selections on μ_p in C''.
    By a similar argument, we can observe $D[\mathcal{O}_{G'}] \Rightarrow \xrightarrow{\overline{\mu_p} \triangleleft j} \Rightarrow \xrightarrow{\overline{\mu_p}[y]} \Rightarrow D''[\mathcal{O}_{G'_j}] \Rightarrow Q_2$, i.e., $E[\mathcal{O}] \overset{\tilde{\alpha}}{\Rightarrow} D[\mathcal{O}_{G'}] \xrightarrow{\overline{\mu_p} \triangleleft j \, \overline{\mu_p}[y]} D''[\mathcal{O}_{G'_j}] \Rightarrow Q_2$. Also in this case, there are no outputs or selections on μ_p in D''.

By assumption and definition,

$$(C''[[\![G']\!]_p], D''[\mathcal{O}_{G'}]) \in \mathbb{B}_0 = \langle G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_k} G', (E[\mathcal{H}_G], E[\mathcal{O}]), p \rangle.$$

Hence, by definition, $(P_2, Q_2) \in \langle G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_k} G' \xrightarrow{s \rangle p:j\langle S_j \rangle} G'_j, (E[\mathcal{H}_G], E[\mathcal{O}]), p \rangle$.

– Take any $(P, Q) \in \langle G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_k} G' \xrightarrow{s \rangle p:j\langle S_j \rangle} G'_j, (E[\mathcal{H}_G], E[\mathcal{O}]), p \rangle$. By definition, there are $(P', Q') \in \langle G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_k} G', (E[\mathcal{H}_G], E[\mathcal{O}]), p \rangle$ such that $P' \xrightarrow{\overline{\mu_p} \triangleleft j \overline{\mu_p}[y]} C[[\![G']\!]_p] \Rightarrow P$ and $Q' \xrightarrow{\overline{\mu_p} \triangleleft j \overline{\mu_p}[y]} D[\mathcal{O}_{G'}] \Rightarrow Q$. Since, $\mathbb{B}_0 = \langle G \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_k} G', (E[\mathcal{H}_G], E[\mathcal{O}]), p \rangle$, by definition $(P, Q) \in \mathbb{B}_2^j$. □

5. Routers in action

We demonstrate our router-based analysis of global types by means of several examples. First, in §5.1 and §5.2 we consider two simple protocols: they illustrate the different components of our approach, and our support for delegation and interleaving. Then in §5.3 we revisit the authorization protocol G_{auth} from Section 1 to illustrate how our analysis supports also more complex protocols featuring also non-local choices and recursion.

5.1. Delegation and interleaving

We illustrate our analysis by considering a global type with delegation and interleaving, based on an example by Toninho and Yoshida [53, Ex. 6.9]. Consider the global type:

$$G_{\mathsf{intrl}} := p \twoheadrightarrow q : 1\langle !\mathsf{int} \,.\, \bullet \rangle \,.\, r \twoheadrightarrow t : 2\langle \mathsf{int} \rangle \,.\, p \twoheadrightarrow q : 3 \,.\, \bullet$$

Following Toninho and Yoshida [53], we define implementations of the roles of the four participants (p, q, r, t) of G_{intrl} using three processes (P_1, P_2, and P_3): P_2 and P_3 implement the roles of q and r, respectively, and P_1 interleaves the roles of p and t by sending a channel s to q and receiving an int value v from r, which it should forward to q over s.

$$P_1 := \overline{p_\mu} \triangleleft 1 \cdot \overline{p_\mu}[s] \cdot (t_\mu \triangleright \{2 : t_\mu(v) \,.\, \overline{s}[w] \cdot v \leftrightarrow w\} \mid \overline{p_\mu} \triangleleft 3 \cdot \overline{p_\mu}[z] \cdot \mathbf{0})$$
$$\vdash p_\mu : \oplus^0\{1 : (\!|!\mathsf{int} \,.\, \bullet|\!) \otimes^1 \oplus^8\{3 : \bullet \otimes^9 \bullet\}\},\ t_\mu : \&^6\{2 : \bullet \parr^7 \bullet\}$$

$$P_2 := q_\mu \triangleright \{1 : q_\mu(y) \,.\, y(x) \,.\, q_\mu \triangleright \{3 : q_\mu(u) \,.\, \mathbf{0}\}\} \vdash q_\mu : \&^2\{1 : \overline{(\!|!\mathsf{int} \,.\, \bullet|\!)} \parr^3 \&^{10}\{3 : \bullet \parr^{11} \bullet\}\}$$

$$P_3 := \overline{r_\mu} \triangleleft 2 \cdot \overline{r_\mu}[\mathbf{33}] \cdot \mathbf{0} \vdash r_\mu : \oplus^4\{2 : \bullet \otimes^5 \bullet\}$$

where '**33**' denotes a closed channel endpoint representing the number "33".

To prove that P_1, P_2, and P_3 correctly implement G_{intrl}, we compose them with the routers synthesized from G_{intrl}. For example, the routers for p and t, to which P_1 will connect, are as follows (omitting curly braces for branches on a single label):

$$\mathcal{R}_p = \mu_p \triangleright 1 \,.\, p_q \triangleleft 1 \cdot \mu_p(s) \,.\, \overline{p_q}[s'] \cdot (s \leftrightarrow s' \mid \mu_p \triangleright 3 \,.\, p_q \triangleleft 3 \cdot \mu_p(z) \,.\, \overline{p_q}[z'] \cdot (z \leftrightarrow z' \mid \mathbf{0}))$$
$$\mathcal{R}_t = t_r \triangleright 2 \,.\, \mu_t \triangleleft 2 \cdot t_r(v) \,.\, \overline{\mu_t}[v'] \cdot (v \leftrightarrow v' \mid \mathbf{0})$$

We assign values to the priorities in $(\!|!\mathsf{int} \,.\, \bullet|\!) = \bullet \otimes^{\mathsf{o}} \bullet$ to ensure that P_1 and P_2 are well-typed; assigning $\mathsf{o} = 8$ works, because the output on s in P_1 occurs after the input on t_μ (which has priority 6–7) and the input on y in P_2 occurs before the second input on q_μ (which has priority 10–11).

The types assigned to p_μ and t_μ in P_1 coincide with $(G_{\mathsf{intrl}} \upharpoonright^0 p)$ and $(G_{\mathsf{intrl}} \upharpoonright^0 t)$, respectively (cf. Definition 23). Therefore, by Theorem 11, the process P_1 connect to the routers for p and t $(\boldsymbol{\nu} p_\mu \mu_p)(\boldsymbol{\nu} t_\mu \mu t)(P_1 \mid \mathcal{R}_p \mid \mathcal{R}_t)$ is well-typed. Similarly, $(\boldsymbol{\nu} q_\mu \mu_q)(P_2 \mid \mathcal{R}_q)$ and $(\boldsymbol{\nu} r_\mu \mu_r)(P_3 \mid \mathcal{R}_r)$ are well-typed.

The composition of these routed implementations results in the following network:

$$N_{\mathsf{intrl}} := \begin{array}{c}(\boldsymbol{\nu} p_q q_p)(\boldsymbol{\nu} p_r r_p)\\(\boldsymbol{\nu} p_t t_p)(\boldsymbol{\nu} q_r r_q)\\(\boldsymbol{\nu} q_t t_q)(\boldsymbol{\nu} r_t t_r)\end{array} \left(\begin{array}{l}(\boldsymbol{\nu} p_\mu \mu_p)(\boldsymbol{\nu} t_\mu \mu_t)(P_1 \mid \mathcal{R}_p \mid \mathcal{R}_t)\\ \mid (\boldsymbol{\nu} q_\mu \mu_q)(P_2 \mid \mathcal{R}_q)\\ \mid (\boldsymbol{\nu} r_\mu \mu_r)(P_3 \mid \mathcal{R}_r)\end{array} \right)$$

We have $N_{\mathsf{intrl}} \in \mathsf{net}(G_{\mathsf{intrl}})$ (cf. Definition 26), so, by Theorem 18, N_{intrl} is deadlock-free and, by Theorem 19 and Theorem 23, it correctly implements G_{intrl}.

5.2. Another example of delegation

Here, we further demonstrate our support for interleaving, showing how a participant can delegate the rest of its interactions in a protocol. The following global type formalizes a protocol in which a Client (c) asks an online Password Manager (p) to login with a Server (s):

$$G_{\mathsf{deleg}} := c \twoheadrightarrow p : \mathsf{login}\langle S\rangle \,.\, G'_{\mathsf{deleg}}$$

where

$$\begin{aligned} S &:= !(?\mathsf{bool} \,.\, \bullet) \,.\, S' \\ S' &:= \&\{\mathsf{passwd} : ?\mathsf{str} \,.\, \oplus\{\mathsf{auth} : !\mathsf{bool} \,.\, \bullet\}\} \\ G'_{\mathsf{deleg}} &:= c \twoheadrightarrow s : \mathsf{passwd}\langle \mathsf{str}\rangle \,.\, s \twoheadrightarrow c : \mathsf{auth}\langle \mathsf{bool}\rangle \,.\, \bullet \end{aligned}$$

Here S' expresses the type of $\mathcal{R}_c$'s channel endpoint μ_c. This means that we can give implementations for c and p such that c can send its channel endpoint c_μ to p, after which p logs in with s in c's place, forwarding the authorization boolean received from s to c. Giving such implementations is relatively straightforward, demonstrating the flexibility of our global types and analysis using APCP and routers.

Using local projection, we can compute a type for c's implementation to safely connect with its router

$$G_{\mathsf{deleg}} \upharpoonright^0 c = \oplus^0\{\mathsf{login} : (\!|S|\!) \otimes^1 (G'_{\mathsf{deleg}} \upharpoonright^4 c)\}$$

where

$$\begin{aligned} (\!|S|\!) &= (\bullet \parr^{\mathsf{o}} \bullet) \otimes^{\kappa} (\!|S'|\!) \\ (\!|S'|\!) &= \&^{\pi}\{\mathsf{passwd} : \bullet \parr^{\rho} \oplus^{\delta}\{\mathsf{auth} : \bullet \otimes^{\phi} \bullet\}\} \\ G'_{\mathsf{deleg}} \upharpoonright^4 c &= \oplus^4\{\mathsf{passwd} : \bullet \otimes^5 \&^{10}\{\mathsf{auth} : \bullet \parr^{11} \bullet\}\} \end{aligned}$$

Notice how $\overline{(\!|S'|\!)} = G'_{\mathsf{deleg}} \upharpoonright^4 c$, given the assignments $\pi = 4, \rho = 5, \delta = 10, \phi = 11$.

We can use these types to guide the design of a process implementation for c. Consider the process:

$$Q := \overline{c_\mu} \triangleleft \mathsf{login} \cdot \overline{c_\mu}[u] \cdot \overline{u}[v] \cdot (u \leftrightarrow c_\mu \mid v(a) \,.\, \mathbf{0}) \vdash \emptyset;\, c_\mu : G_{\mathsf{deleg}} \upharpoonright^0 c$$

This implementation is interesting: after the first exchange in G_{deleg}—sending a fresh channel u (to p)—c sends another fresh channel v over u; then, c delegates the rest of its exchanges in G'_{deleg} by forwarding all traffic on c_μ over u; in the meantime, c awaits an authorization boolean over v.

Again, using local projection, we can compute a type for p's implementation to connect with its router:

$$G_{\mathsf{deleg}} \upharpoonright^0 p = \&^2\{\mathsf{login} : \overline{(\!|S|\!)} \parr^3 \bullet\}$$

We can then use it to type the following implementation for p:

$$P := p_\mu \triangleright \left\{ \begin{array}{l} \mathsf{login} : p_\mu(c_\mu) \,.\, c_\mu(v) \\ \qquad \cdot \overline{c_\mu} \triangleleft \mathsf{passwd} \cdot \overline{c_\mu}[\boldsymbol{pwd123}] \\ \qquad \cdot c_\mu \triangleleft \{\mathsf{auth} : c_\mu(a) \,.\, \overline{v}[a'] \cdot a \leftrightarrow a'\} \end{array} \right\} \vdash \emptyset;\, p_\mu : G_{\mathsf{deleg}} \upharpoonright^0 p$$

In this implementation, p receives a channel c_μ (from c) over which it first receives a channel v. Then, it behaves over c_μ according to c's role in G'_{deleg}. Finally, p forwards the authorization boolean received from s over v, effectively sending the boolean to c.

Given an implementation for s, say $S \vdash \emptyset;\, s_\mu : G_{\mathsf{deleg}} \upharpoonright^0 s$, what remains is to assign values to the remaining priorities in $(\!|S|\!)$: assigning $\mathsf{o} = 12, \kappa = 4$ works. Now, we can compose the implementations P, Q and S with their respective routers and then compose these routed implementations together to form a deadlock-free network of G_{deleg}. This way, e.g., the router for c is as follows (again, omitting curly braces for branches on a single label):

$$\begin{aligned} \mathcal{R}_c = \mu_c \triangleright \mathsf{login} \,.\, c_p \triangleleft \mathsf{login} \cdot \mu_c(u) \,.\, \overline{c_p}[u'] \cdot (\\ \quad u \leftrightarrow u' \mid \mu_c \triangleright \mathsf{passwd} \,.\, c_s \triangleleft \mathsf{passwd} \cdot \mu_c(v) \,.\, \overline{c_s}[v'] \cdot (\\ \quad\quad v \leftrightarrow v' \mid c_s \triangleright \mathsf{auth} \,.\, \mu_c \triangleleft \mathsf{auth} \cdot c_s(w) \,.\, \overline{\mu_c}[w'] \cdot (w \leftrightarrow w' \mid \mathbf{0}))) \end{aligned}$$

Interestingly, the router is agnostic of the fact that the endpoint u it receives over μ_c is in fact the opposite endpoint of the channel formed by μ_c.

5.3. The authorization protocol in action

Let us repeat G_{auth} from Section 1:

$$G_{\mathsf{auth}} = \mu X \,.\, s \twoheadrightarrow c \left\{ \begin{array}{l} \mathsf{login} \,.\, c \twoheadrightarrow a : \mathsf{passwd}\langle \mathsf{str}\rangle \,.\, a \twoheadrightarrow s : \mathsf{auth}\langle \mathsf{bool}\rangle \,.\, X, \\ \mathsf{quit} \,.\, c \twoheadrightarrow a : \mathsf{quit} \,.\, \bullet \end{array} \right\}$$

The relative projections of G_{auth} are as follows:

$$G_{\mathsf{auth}} \upharpoonright (s,a) = \mu X \,.\, s!c \left\{ \begin{array}{l} \mathsf{login} \,.\, \mathsf{skip} \,.\, a : \mathsf{auth}\langle \mathsf{bool}\rangle \,.\, X, \\ \mathsf{quit} \,.\, \mathsf{skip} \,.\, \bullet \end{array} \right\}$$

$$G_{\mathsf{auth}} \upharpoonright (c,a) = \mu X \,.\, c?s \left\{ \begin{array}{l} \mathsf{login} \,.\, c : \mathsf{passwd}\langle \mathsf{str}\rangle \,.\, \mathsf{skip} \,.\, X, \\ \mathsf{quit} \,.\, c : \mathsf{quit} \,.\, \bullet \end{array} \right\}$$

$$G_{\mathsf{auth}} \upharpoonright (s,c) = \mu X \,.\, s \left\{ \begin{array}{l} \mathsf{login} \,.\, \mathsf{skip}^2 \,.\, X, \\ \mathsf{quit} \,.\, \mathsf{skip} \,.\, \bullet \end{array} \right\}$$

$$\mathcal{R}_c = \mu X(\mu_c, c_s, c_a) \,.\, c_s \triangleright \left\{ \begin{array}{l} \mathsf{login} : \overline{\mu_c} \lhd \mathsf{login} \cdot \overline{c_a} \lhd \mathsf{login} \cdot c_s(u) \,.\, \overline{\mu_c}[u'] \\ \qquad \cdot (u \leftrightarrow u' \mid \mu_c \triangleright \left\{ \begin{array}{l} \mathsf{passwd} : \overline{c_a} \lhd \mathsf{passwd} \cdot \mu_c(v) \,.\, \overline{c_a}[v'] \\ \qquad \cdot (v \leftrightarrow v' \mid X\langle \mu_c, c_s, c_a\rangle) \end{array} \right\}), \\ \mathsf{quit} \;\; : \overline{\mu_c} \lhd \mathsf{quit} \cdot \overline{c_a} \lhd \mathsf{quit} \cdot c_s(w) \,.\, \overline{\mu_c}[w'] \\ \qquad \cdot (w \leftrightarrow w' \mid \mu_c \triangleright \left\{ \begin{array}{l} \mathsf{quit} : \overline{c_a} \lhd \mathsf{quit} \cdot \mu_c(z) \,.\, \overline{c_a}[z'] \\ \qquad \cdot (z \leftrightarrow z' \mid \mathbf{0}) \end{array} \right\}) \end{array} \right\}$$

$$\vdash \mu_c : \mu X \,.\, \oplus^2 \left\{ \begin{array}{l} \mathsf{login} : \bullet \otimes^3 \&^4 \{\mathsf{passwd} : \bullet \,\aleph^5\, X\}, \\ \mathsf{quit} \;\; : \bullet \otimes^3 \&^4 \{\mathsf{quit} : \bullet \,\aleph^5\, \bullet\} \end{array} \right\} = \overline{(G_{\mathsf{auth}} \upharpoonright {}^0\, c)},$$

$$c_s : \mu X \,.\, \&^1 \{\mathsf{login} : \bullet \,\aleph^2\, X, \mathsf{quit} : \bullet \,\aleph^2\, \bullet\} = (\!| G_{\mathsf{auth}} \upharpoonright (c,s) |\!)^0_{c\rangle s},$$

$$c_a : \mu X \,.\, \oplus^2 \left\{ \begin{array}{l} \mathsf{login} : \oplus^5 \{\mathsf{passwd} : \bullet \otimes^6 X\}, \\ \mathsf{quit} \;\; : \oplus^5 \{\mathsf{quit} : \bullet \otimes^6 \bullet\} \end{array} \right\} = (\!| G_{\mathsf{auth}} \upharpoonright (c,a) |\!)^0_{c\rangle a}$$

$$\mathcal{R}_s = \mu X(\mu_s, s_c, s_a) \,.\, \mu_s \triangleright \left\{ \begin{array}{l} \mathsf{login} : \overline{s_c} \lhd \mathsf{login} \cdot \overline{s_a} \lhd \mathsf{login} \cdot \mu_s(u) \,.\, \overline{s_c}[u'] \\ \qquad \cdot (u \leftrightarrow u' \mid s_a \triangleright \left\{ \begin{array}{l} \mathsf{auth} : \overline{\mu_s} \lhd \mathsf{auth} \cdot s_a(v) \,.\, \overline{\mu_s}[v'] \\ \qquad \cdot (v \leftrightarrow v' \mid X\langle \mu_s, s_c, s_a\rangle) \end{array} \right\}), \\ \mathsf{quit} \;\; : \overline{s_c} \lhd \mathsf{quit} \cdot \overline{s_a} \lhd \mathsf{quit} \cdot \mu_s(v) \,.\, \overline{s_c}[v'] \\ \qquad \cdot (v \leftrightarrow v' \mid \mathbf{0}) \end{array} \right\}$$

$$\vdash \mu_s : \mu X \,.\, \&^0 \{\mathsf{login} : \bullet \,\aleph^1\, \oplus^{10} \{\mathsf{auth} : \bullet \otimes^{11} X\}, \mathsf{quit} : \bullet \,\aleph^1\, \bullet\} = \overline{(G_{\mathsf{auth}} \upharpoonright {}^0\, s)},$$

$$s_c : \mu X \,.\, \oplus^1 \{\mathsf{login} : \bullet \otimes^2 X, \mathsf{quit} : \bullet \otimes^2 \bullet\} = (\!| G_{\mathsf{auth}} \upharpoonright (s,c) |\!)^0_{s\rangle c},$$

$$s_a : \mu X \,.\, \oplus^1 \{\mathsf{login} : \&^9 \{\mathsf{auth} : \bullet \,\aleph^{10}\, X\}, \mathsf{quit} : \bullet\} = (\!| G_{\mathsf{auth}} \upharpoonright (s,a) |\!)^0_{s\rangle a}$$

$$\mathcal{R}_a = \mu X(\mu_a, a_c, a_s) \,.\, a_s \triangleright \left\{ \begin{array}{l} \mathsf{login} : \overline{\mu_a} \lhd \mathsf{login} \\ \cdot\, a_c \triangleright \left\{ \begin{array}{l} \mathsf{login} : \\ \quad a_c \triangleright \left\{ \begin{array}{l} \mathsf{passwd} : \overline{\mu_a} \lhd \mathsf{passwd} \cdot a_c(u) \,.\, \overline{\mu_a}[u'] \\ \quad \cdot \left(\begin{array}{l} u \leftrightarrow u' \\ \quad \mid \mu_a \triangleright \left\{ \begin{array}{l} \mathsf{auth} : \\ \overline{a_s} \lhd \mathsf{auth} \cdot \mu_a(v) \,.\, \overline{a_s}[v'] \\ \cdot (v \leftrightarrow v' \mid X\langle \mu_a, a_c, a_s\rangle) \end{array} \right\} \end{array} \right) \end{array} \right\}, \\ \quad \mathsf{quit} : \mathsf{alarm}(\mu_a, a_c, a_s) \end{array} \right\}, \\ \mathsf{quit} : \overline{\mu_a} \lhd \mathsf{quit} \\ \cdot\, a_c \triangleright \left\{ \begin{array}{l} \mathsf{login} : \mathsf{alarm}(\mu_a, a_c, a_s), \\ \mathsf{quit} : a_c \triangleright \{\mathsf{quit} : \overline{\mu_a} \lhd \mathsf{quit} \cdot a_c(w) \,.\, \overline{\mu_a}[w'] \cdot (w \leftrightarrow w' \mid \mathbf{0})\}, \end{array} \right\} \end{array} \right\}$$

$$\vdash \mu_a : \mu X \,.\, \oplus^2 \left\{ \begin{array}{l} \mathsf{login} : \oplus^6 \{\mathsf{passwd} : \bullet \otimes^7 \&^8 \{\mathsf{auth} : \bullet \,\aleph^9\, X\}\}, \\ \mathsf{quit} \;\; : \oplus^6 \{\mathsf{quit} : \bullet \otimes^7 \bullet\} \end{array} \right\} = \overline{(G_{\mathsf{auth}} \upharpoonright {}^0\, a)},$$

$$a_c : \mu X \,.\, \&^2 \left\{ \begin{array}{l} \mathsf{login} : \&^5 \{\mathsf{passwd} : \bullet \,\aleph^6\, X\}, \\ \mathsf{quit} \;\; : \&^5 \{\mathsf{quit} : \bullet \,\aleph^6\, \bullet\} \end{array} \right\} = (\!| G_{\mathsf{auth}} \upharpoonright (a,c) |\!)^0_{a\rangle c},$$

$$a_s : \mu X \,.\, \&^1 \{\mathsf{login} : \oplus^9 \{\mathsf{auth} : \bullet \otimes^{10} X\}, \mathsf{quit} : \bullet\} = (\!| G_{\mathsf{auth}} \upharpoonright (a,s) |\!)^0_{a\rangle s}$$

Fig. 14. Routers synthesized from G_{auth}.

The typed routers synthesized from G_{auth} are given in Fig. 14. Let us explain the behavior of $\mathcal{R}_a$, the router of a. $\mathcal{R}_a$ is a recursive process on recursion variable X, using the endpoint for the implementation μ_a and the endpoint for the other routers a_c and a_s as context. The initial message in G_{auth} from s to c is a dependency for a's interactions with both s and

$$
\begin{aligned}
&\mathsf{O}_{\{c,s,a\}}[G_{\mathsf{auth}}] \\
&= \mu X(\mu_c, \mu_s, \mu_a) \\
&\quad . \, \mu_s \triangleright \left\{ \begin{array}{l}
\mathsf{login} : \overline{\mu_c} \triangleleft \mathsf{login} \cdot \overline{\mu_a} \triangleleft \mathsf{login} \cdot \mu_s(u) \, . \, \overline{\mu_c}[u'] \\
\qquad \cdot (u \leftrightarrow u' \mid \mu_c \triangleright \left\{ \begin{array}{l} \mathsf{auth} : \overline{\mu_a} \triangleleft \mathsf{passwd} \cdot \mu_c(v) \, . \, \overline{\mu_a}[v'] \\ \qquad \cdot (v \leftrightarrow v' \mid \mu_a \triangleright \left\{ \begin{array}{l} \mathsf{auth} : \overline{\mu_s} \triangleleft \mathsf{auth} \cdot \mu_a(w) \, . \, \overline{\mu_s}[w'] \\ \qquad \cdot (w \leftrightarrow w' \mid X\langle \mu_c, \mu_s, \mu_a \rangle) \end{array} \right\}) \end{array} \right\}), \\
\mathsf{quit} \ : \overline{\mu_c} \triangleleft \mathsf{quit} \cdot \overline{\mu_a} \triangleleft \mathsf{quit} \cdot \mu_s(z) \, . \, \overline{\mu_c}[z'] \\
\qquad \cdot (z \leftrightarrow z' \mid \mu_c \triangleright \{\mathsf{quit} : \overline{\mu_a} \triangleleft \mathsf{quit} \cdot \mu_c(y) \, . \, \overline{\mu_a}[y'] \cdot (y \leftrightarrow y' \mid \mathbf{0})\})
\end{array} \right. \\
&\vdash \mu_c : \overline{G_{\mathsf{auth}} \upharpoonright^0 c}, \mu_s : \overline{G_{\mathsf{auth}} \upharpoonright^0 s}, \mu_a : \overline{G_{\mathsf{auth}} \upharpoonright^0 a}
\end{aligned}
$$

Fig. 15. Orchestrator synthesized from G_{auth} (cf. Definition 36).

c. Therefore, the router first branches on the first dependency with s: a label received over a_s (login or quit). Let us detail the login branch. Here, the router sends login over μ_a. Then, the router branches on the second dependency with c: a label received over a_c (again, login or quit).

- In the second login branch, the router receives the label passwd over a_c, which it then sends over μ_a. The router then receives an endpoint (the password) over a_c, which it forwards over μ_a. Finally, the router receives the label auth over μ_a, which it sends over a_s. Then, the router receives an endpoint (the authorization result) over μ_a, which it forwards over a_s. The router then recurses to the beginning of the loop on the recursion variable X, passing the endpoints μ_a, a_s, a_c as recursive context.
- In the quit branch, the router is in an inconsistent state, because it has received a label over a_c which does not concur with the label received over a_s. Hence, the router signals an alarm on its endpoints μ_a, a_s, a_c.

Notice how the typing of the routers in Fig. 14 follows Theorem 11: for each $p \in \{c, s, a\}$, the endpoint μ_p is typed with local projection (Definition 23), and for each $q \in \{c, s, a\} \setminus \{p\}$ the endpoint p_q is typed with relative projection (Definitions 17 and 24).

Consider again the participant implementations given in Example 1: P implements the role of c, Q the role of s, and R the role of a. Notice that the types of the channels of these processes coincide with relative projections:

$$P \vdash \emptyset; c_\mu : G_{\mathsf{auth}} \upharpoonright^0 c \qquad Q \vdash \emptyset; s_\mu : G_{\mathsf{auth}} \upharpoonright^0 s \qquad R \vdash \emptyset; a_\mu : G_{\mathsf{auth}} \upharpoonright^0 a$$

Let us explore how to compose these implementations with their respective routers. The order of composition determines the network topology.

Decentralized By first composing each router with their respective implementation, and then composing the resulting routed implementations, we obtain a decentralized topology:

$$N_{\mathsf{auth}}^{\mathsf{decentralized}} := \begin{array}{c} (\boldsymbol{\nu} c_s s_c) \\ (\boldsymbol{\nu} c_a a_c) \\ (\boldsymbol{\nu} s_a a_s) \end{array} \left(\begin{array}{r} (\boldsymbol{\nu} \mu_c c_\mu)(\mathcal{R}_c \mid P) \\ \mid (\boldsymbol{\nu} \mu_s s_\mu)(\mathcal{R}_s \mid Q) \\ \mid (\boldsymbol{\nu} \mu_a a_\mu)(\mathcal{R}_a \mid R) \end{array} \right)$$

This composition is in fact a network of routed implementations of G (cf. Definition 26), so Theorems 18, 19 and 23 apply: we have $N_{\mathsf{auth}}^{\mathsf{decentralized}} \in \mathsf{net}(G_{\mathsf{auth}})$, so $N_{\mathsf{auth}}^{\mathsf{decentralized}}$ behaves as specified by G_{auth} and is deadlock-free.

Centralized By first composing the routers, and then composing the connected routers with each implementation, we obtain a centralized topology:

$$N_{\mathsf{auth}}^{\mathsf{centralized}} := \begin{array}{c} (\boldsymbol{\nu} \mu_c c_\mu) \\ (\boldsymbol{\nu} \mu_s s_\mu) \\ (\boldsymbol{\nu} \mu_a a_\mu) \end{array} \left(\begin{array}{c} (\boldsymbol{\nu} c_s s_c) \\ (\boldsymbol{\nu} c_a a_c) \\ (\boldsymbol{\nu} s_a a_s) \end{array} \left(\begin{array}{c} \mathcal{R}_c \\ \mid \mathcal{R}_s \\ \mid \mathcal{R}_a \end{array} \right) \begin{array}{l} \mid P \\ \mid Q \\ \mid R \end{array} \right)$$

Note that the composition of routers is a *hub of routers* (Definition 37). Consider the composition of P, Q and R with the orchestrator of G_{auth} (given in Fig. 15):

$$N_{\mathsf{auth}}^{\mathsf{orchestrator}} := \begin{array}{c} (\boldsymbol{\nu} \mu_c c_\mu) \\ (\boldsymbol{\nu} \mu_s s_\mu) \\ (\boldsymbol{\nu} \mu_a a_\mu) \end{array} \left(\begin{array}{r} \mid P \\ \mathsf{O}_{\{c,s,a\}}[G_{\mathsf{auth}}] \mid Q \\ \mid R \end{array} \right)$$

By Theorem 27, the hub of routers and the orchestrator of G_{auth} are weakly bisimilar (Definition 39). Hence, $N_{\mathsf{auth}}^{\mathsf{centralized}}$ and $N_{\mathsf{auth}}^{\mathsf{orchestrator}}$ behave the same.

Since each of $N_{\mathsf{auth}}^{\mathsf{top}}$ with $\mathsf{top} \in \{\mathsf{decentralized}, \mathsf{centralized}, \mathsf{orchestrator}\}$ is typable in empty contexts, by Theorem 18, each of these compositions is deadlock-free. Moreover, $N_{\mathsf{auth}}^{\mathsf{decentralized}}$ and $N_{\mathsf{auth}}^{\mathsf{centralized}}$ are structurally congruent, so, by Theorems 19 and 23, they behave as prescribed by G_{auth}. Finally, by Theorem 27, $N_{\mathsf{auth}}^{\mathsf{centralized}}$ and $N_{\mathsf{auth}}^{\mathsf{orchestrator}}$ are bisimilar, and so $N_{\mathsf{auth}}^{\mathsf{orchestrator}}$ also behaves as prescribed by G_{auth}.

6. Related work

Types for deadlock-freedom Our decentralized analysis of global types is related to type systems that ensure deadlock-freedom for multiparty sessions with delegation and interleaving [7,46,22]. Unlike these works, we rely on a type system for *binary* sessions which is simple and enables an expressive analysis of global types. Coppo et al. [7,21,22] give type systems for multiparty protocols, with asynchrony and support for interleaved sessions by tracking of mutual dependencies between them; as per Toninho and Yoshida [53], our example in Section 5.1 is typable in APCP but untypable in their system. Padovani et al. [46] develop a type system that enforces liveness properties for multiparty sessions, defined on top of a π-calculus with labeled communication. Rather than global types, their type structure follows approaches based on *conversation types* [16]. Toninho and Yoshida [53] analyze binary sessions, leveraging on deadlock-freedom results for multiparty sessions to extend Wadler's CLL [57] with cyclic networks. Their process language is synchronous and uses replication rather than recursion. We note that their Examples 6.8 and 6.9 can be typed in APCP (cf. § 5.1); a detailed comparison between their extended CLL and APCP is interesting future work.

MPST and binary analyses of global types There are many works on MPST and their integration into programming languages; see [40,3] for surveys. Triggered by flawed proofs of type safety and limitations of usual theories, Scalas and Yoshida [51] define a meta-framework of multiparty protocols based on local types, without global types and projection. Their work has been a source of inspiration for our developments; we address similar issues by adopting relative types, instead of cutting ties with global types.

As already mentioned, Caires and Pérez [12] and Carbone et al. [17] reduce the analysis of global types to binary session type systems based on intuitionistic and classical linear logic, respectively. Our routers strictly generalize the centralized mediums of Caires and Pérez (cf. § 4.4). We substantially improve over the expressivity of the decentralized approach of Carbone et al. based on coherence, but reliant on encodings into centralized arbiters; for instance, their approach does not support the example from Toninho and Yoshida [53] we discuss in § 5.1. Also, Caires and Pérez support neither recursive global types nor asynchronous communication, and neither do Carbone et al..

Scalas et al. [50] leverage on an encoding of binary session types into *linear types* [24,43] to reduce multiparty sessions to processes typable with linear types, with applications in Scala programming. Their analysis is decentralized but covers processes with synchronous communication only; also, their deadlock-freedom result is limited with respect to ours: it does not support interleaving, such as in the example in § 5.1.

Monitoring through MPST Our work and the works discussed so far all consider the verification of implementations of multiparty protocols through static type checking. Bocchi et al. [8] use a *dynamic* approach: communication between implementations is enacted by *monitors*, which are derived from the global type to prevent protocol violations. In their approach, Bocchi et al. rely on the traditional workflow for MPST: projection onto binary session types based on the merge operation. Interestingly, Bocchi et al.'s semantics relies on *routing*, which is similar in spirit, but not in details, to our routers: their routing approach abstracts away from the actual network structure, while our routers enable the concrete realization of a decentralized network structure. We also note that Bocchi et al.'s monitors, based on finite state machines, live on the level of semantics, while our routers, π-calculus processes, live on the same level as implementations. The theory by Bocchi et al. has resulted in the development of tools for a practical application of monitoring in Python [27], including an extension to real-time systems [45].

Other approaches to multiparty protocols In a broader context, Message Sequence Charts (MSCs) provide graphical specifications of multiparty protocols. Alur et al. [2] and Abdallah et al. [1] study the decidability of model-checking properties such as implementability of MSC Graphs and High-level MSCs (HMSCs) as Communicating FSMs (CFSMs). Genest et al. [34] study the synthesis of implementations of HMSCs as CFSMs; as we do, they use extra synchronization messages in some cases. We follow an entirely different research strand: our analysis is type-based and targets well-formed global types that are implementable by design. We note that the decidability of key notions for MPST (such as well-formedness and typability) has been addressed in [38].

Collaboration diagrams are another visual model for communicating processes (see, e.g. [10]). Salaün et al. [49] encode collaboration diagrams into the LOTOS process algebra [30] to enable model-checking [32], realizability checks for synchronous and asynchronous communication, and synthesis of participant implementations. Their implementation synthesis is reminiscent of our router synthesis, and also adds extra synchronization messages to realize otherwise unrealizable protocols with non-local choices.

7. Conclusion

We have developed a new analysis of multiparty protocols specified as global types. As a distinguishing feature, our analysis accounts for multiparty protocols implemented by arbitrary process networks, which can be centralized (as in orchestration-based approaches) but also decentralized (as in choreography-based approaches). Another salient feature is that we can ensure both protocol conformance (protocol fidelity, communication safety) and deadlock-freedom, which is notoriously hard to establish for protocols/implementations involving delegation and interleaving. To this end, we have considered asynchronous process implementations in APCP, the typed process language that we introduced in [54]. Our analysis enables the transference of correctness properties from APCP to multiparty protocols. We have illustrated these features using the authorization protocol G_{auth} adapted from Scalas and Yoshida [51] as a running example; additional examples further justify how our approach improves over previous analyses (cf. Section 5).

Our analysis of multiparty protocols rests upon three key innovations: *routers*, which enable global type analysis as decentralized networks; *relative types* that capture protocols between pairs of participants; *relative projection*, which admits global types with non-local choices. In our opinion, these notions are interesting on their own. In particular, relative types shed new light on more expressive protocol specifications than usual MPST, which are tied to notions of local types and merge/subtyping.

There are several interesting avenues for future work. Comparing relative and merge-based well-formedness would continue the tread of new projections of global types (cf. App. A for initial findings). We would also like to develop a type system based on relative types, integrating the logic of routers into a static type checking that ensures deadlock-freedom for processes. Finally, we are interested in developing practical tool support based on our findings. For this latter point, following [42], we would like to first formalize a theory of runtime monitoring based on routers, which can already be seen as an elementary form of *choreographed monitoring* (cf. [31]).

CRediT authorship contribution statement

All persons who meet authorship criteria are listed as authors, and all authors certify that they have participated sufficiently in the work to take public responsibility for the content.

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 the anonymous reviewers for their constructive feedback and suggestions, which were enormously helpful to improve the presentation. We are also grateful to Jules Jacobs for valuable comments and suggestions.

This research has been supported by the Dutch Research Council (NWO) under project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).

Appendix A. Comparing merge-based well-formedness and relative well-formedness

It is instructive to examine how the notion of well-formed global types induced by our relative projection compares to *merge-based* well-formedness, the notion induced by (usual) local projection [39,18].

Before we recall the definition of merge-based well-formedness, we define the projection of global types to local types. Local types express one particular participant's perspective of a global protocol. Although skip is not part of standard definitions of local types, we include it to enable a fair comparison with relative types.

Definition 42 *(Local types). Local types* L are defined as follows, where the S_i are the message types from Definition 12:

$$L ::= ?p\{i\langle S\rangle . L\}_{i\in I} \mid !p\{i\langle S\rangle . L\}_{i\in I} \mid \mu X . L \mid X \mid \bullet \mid \mathsf{skip} . L$$

The local types $?p\{i\langle S_i\rangle . L_i\}_{i\in I}$ and $!p\{i\langle S_i\rangle . L_i\}_{i\in I}$ represent receiving a choice from p and sending a choice to p, respectively. All of $\bullet$, $\mu X . L$, X, and skip are just as before.

Instead of external dependencies, the projection onto local types relies on an operation on local types called *merge*. Intuitively, merge allows combining overlapping but not necessarily identical receiving constructs. This is one main difference with respect to our relative projection.

Definition 43 *(Merge of local types).* For local types L_1 and L_2, we define $L_1 \sqcup L_2$ as the *merge* of L_1 and L_2:

$$\mathsf{skip} . L_1 \sqcup \mathsf{skip} . L_2 := L_1 \sqcup L_2 \qquad\qquad \bullet \sqcup \bullet := \bullet$$

$$\mu X \,.\, L_1 \sqcup \mu X \,.\, L_2 := \mu X \,.\, (L_1 \sqcup L_2) \qquad X \sqcup X := X$$

$$!p\{i\langle S_i\rangle \,.\, L_i\}_{i\in I} \sqcup !p\{i\langle S_i\rangle \,.\, L_i\}_{i\in I} := !p\{i\langle S_i\rangle \,.\, L_i\}_{i\in I}$$

$$?p\{i\langle S_i\rangle \,.\, L_i\}_{i\in I} \sqcup ?p\{j\langle S'_j\rangle \,.\, L'_j\}_{j\in J} := ?p\begin{pmatrix} \{i\langle S_i\rangle \,.\, L_i\}_{i\in I\setminus J} \\ \cup\, \{j\langle S'_j\rangle \,.\, L'_j\}_{j\in J\setminus I} \\ \cup\, \{k\langle S_k \sqcup S'_k\rangle \,.\, (L_k \sqcup L'_k)\}_{k\in I\cap J}\end{pmatrix}$$

The merge between message types $S_1 \sqcup S_2$ corresponds to the identity function. If the local types do not match the above definition, their merge is undefined.

We can now define local projection based on merge:

Definition 44 *(Merge-based local projection).* For global type G and participant p, we define $G \upharpoonright p$ as the *merge-based local projection* of G under p:

$$\bullet \upharpoonright p := \bullet \qquad (\mathsf{skip} \,.\, G) \upharpoonright p := \mathsf{skip} \,.\, (G \upharpoonright p) \qquad X \upharpoonright p := X$$

$$(\mu X \,.\, G) \upharpoonright p := \begin{cases} \bullet & \text{if } G \upharpoonright p = \mathsf{skip}^* \,.\, \bullet \text{ or } G \upharpoonright p = \mathsf{skip}^* \,.\, X \\ \mu X \,.\, (G \upharpoonright p) & \text{otherwise} \end{cases}$$

$$(r \twoheadrightarrow s\{i\langle U_i\rangle \,.\, G_i\}_{i\in I}) \upharpoonright p := \begin{cases} ?r\{i\langle U_i\rangle \,.\, (G_i \upharpoonright p)\}_{i\in I} & \text{if } p = s \\ !s\{i\langle U_i\rangle \,.\, (G_i \upharpoonright p)\}_{i\in I} & \text{if } p = r \\ \mathsf{skip} \,.\, (\sqcup_{i\in I}(G_i \upharpoonright p)) & \text{otherwise} \end{cases}$$

$$(G_1 \mid G_2) \upharpoonright p := \begin{cases} G_1 \upharpoonright p & \text{if } p \in \mathsf{prt}(G_1) \text{ and } p \notin \mathsf{prt}(G_2) \\ G_2 \upharpoonright p & \text{if } p \in \mathsf{prt}(G_2) \text{ and } p \notin \mathsf{prt}(G_1) \\ \bullet & \text{if } p \notin \mathsf{prt}(G_1) \cup \mathsf{prt}(G_2) \end{cases}$$

Definition 45 *(Merge well-formedness).* A global type G is *merge well-formed* if, for every $p \in \mathsf{prt}(G)$, the merge-based local projection $G \upharpoonright p$ is defined.

The classes of relative and merge-based well-formed global types overlap: there are protocols that can be expressed using dependencies in relative types, as well as using merge in local types. Interestingly, the classes are *incomparable*: some relative well-formed global types are not merge-based well-formed, and vice versa. We now explore these differences.

A.1. Relative well-formed, not merge well-formed

The merge of local types with outgoing messages of different labels is undefined. Therefore, if a global type has communications, e.g., from s to a with different labels across branches of a prior communication between b and a, the global type is not merge well-formed. In contrast, such global types can be relative well-formed, because the prior communication may induce a dependency. Similarly, global types with communications with different participants across branches of a prior communication are never merge well-formed, but may be relative well-formed. The following example demonstrates a global type with messages of different labels across branches of a prior communication:

Example. We give an adaptation of the two-buyer-seller protocol in which Seller (s) tells Alice (a) to pay or not, depending on whether Bob (b) tells a to buy or not.

$$G_{\mathsf{rwf}} := b \twoheadrightarrow a \begin{Bmatrix} \mathsf{ok} \,.\, s \twoheadrightarrow a : \mathsf{pay}\langle\mathsf{int}\rangle \,.\, \bullet, \\ \mathsf{cancel} \,.\, s \twoheadrightarrow a : \mathsf{cancel} \,.\, \bullet \end{Bmatrix}$$

This protocol is relative well-formed, as the relative projections under every combination of participants are defined. Notice how there is a dependency in the relative projection under s and a:

$$G_{\mathsf{rwf}} \upharpoonright (s, a) = a?b \begin{Bmatrix} \mathsf{ok} \,.\, s : \mathsf{pay}\langle\mathsf{int}\rangle \,.\, \bullet, \\ \mathsf{cancel} \,.\, s : \mathsf{cancel} \,.\, \bullet \end{Bmatrix}$$

However, we do not have merge well-formedness: the merge-based local projection under s is not defined:

$$G_{\mathsf{rwf}} \upharpoonright s = \mathsf{skip} \,.\, (!a : \mathsf{pay}\langle\mathsf{int}\rangle \,.\, \bullet \sqcup\, !a : \mathsf{cancel} \,.\, \bullet)$$

A.2. Merge well-formed, not relative well-formed

For a communication between, e.g., a and b to induce a dependency for subsequent communications between other participants, at least one of a and b must be involved. Therefore, global types where communications with participants other than a and b have different labels across branches of a prior communication between a and b are never relative well-formed. In contrast, merge can combine the reception of different labels, so such global types may be merge well-formed—as long as the sender is aware of which branch has been taken before. The following example demonstrates such a situation, and explains how such global types can be modified to be relative well-formed:

Example. Consider a variant of the two-buyer-seller protocol in which Seller (s) invokes a new participant, Mail-service (m), to deliver the requested product. In the following global type, Bob (b) tells Alice (a) of its decision to buy or not, after which b sends the same choice to s, who then either invokes m to deliver the product or not:

$$G_{\mathsf{mwf}} := b \twoheadrightarrow a \begin{Bmatrix} \mathsf{ok} \,.\, b \twoheadrightarrow s : \mathsf{ok} \,.\, s \twoheadrightarrow m : \mathsf{deliver}\langle \mathsf{str} \rangle \,.\, \bullet, \\ \mathsf{quit} \,.\, b \twoheadrightarrow s : \mathsf{quit} \,.\, s \twoheadrightarrow m : \mathsf{quit} \,.\, \bullet \end{Bmatrix}$$

G_{mwf} is merge well-formed: the merge-based local projections under all participants are defined. Notice how the two different messages from s are merged in the merge-based local projection under m:

$$G_{\mathsf{mwf}} \upharpoonright m = \mathsf{skip}^2 \,.\, ?s \begin{Bmatrix} \mathsf{deliver}\langle \mathsf{str} \rangle \,.\, \bullet, \\ \mathsf{quit} \,.\, \bullet \end{Bmatrix}$$

G_{mwf} is not relative well-formed: the relative projection under s and m is not defined. The initial exchange between b and a cannot induce a dependency, since neither of s and m is involved. Hence, the relative projections of both branches must be identical, but they are not:

$$\mathsf{skip} \,.\, s : \mathsf{deliver}\langle \mathsf{str} \rangle \,.\, \bullet \neq \mathsf{skip} \,.\, s : \mathsf{quit} \,.\, \bullet$$

We recover relative well-formedness by modifying G_{mwf}: we give s the same options to send to m in both branches of the initial communication:

$$G'_{\mathsf{mwf}} := b \twoheadrightarrow a \begin{Bmatrix} \mathsf{ok} \,.\, b \twoheadrightarrow s : \mathsf{ok} \,.\, s \twoheadrightarrow m\{\mathsf{deliver}\langle \mathsf{str} \rangle \,.\, \bullet, \quad \mathsf{quit} \,.\, \bullet\}, \\ \mathsf{quit} \,.\, b \twoheadrightarrow s : \mathsf{quit} \,.\, s \twoheadrightarrow m\{\mathsf{deliver}\langle \mathsf{str} \rangle \,.\, \bullet, \quad \mathsf{quit} \,.\, \bullet\} \end{Bmatrix}$$

The new protocol is still merge well-formed, but it is now relative well-formed too; the relative projection under s and m is defined:

$$G'_{\mathsf{mwf}} \upharpoonright (s, m) = \mathsf{skip}^2 \,.\, s \begin{Bmatrix} \mathsf{deliver}\langle \mathsf{address} \rangle \,.\, \bullet, \\ \mathsf{quit} \,.\, \bullet \end{Bmatrix}$$

This modification may not be ideal, though, because s can quit the protocol even if b has confirmed the transaction, and that s can still invoke a delivery even if b has quit the transaction.

References

[1] Rouwaida Abdallah, Loïc Hélouët, Claude Jard, Distributed implementation of message sequence charts, Softw. Syst. Model. 14 (2) (May 2015) 1029–1048, https://doi.org/10.1007/s10270-013-0357-1.

[2] Rajeev Alur, Kousha Etessami, Mihalis Yannakakis, Realizability and verification of MSC graphs, Theor. Comput. Sci. 331 (1) (February 2005) 97–114, https://doi.org/10.1016/j.tcs.2004.09.034.

[3] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, Nobuko Yoshida, Behavioral types in programming languages, Found. Trends Program. Lang. 3 (2–3) (July 2016) 95–230, https://doi.org/10.1561/2500000031.

[4] Robert Atkey, Sam Lindley, J. Garrett Morris, Conflation confers concurrency, in: Sam Lindley, Conor McBride, Phil Trinder, Don 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, Springer International Publishing, Cham, 2016, pp. 32–55.

[5] Franco Barbanera, Mariangiola Dezani-Ciancaglini, Open multiparty sessions, Electron. Proc. Theor. Comput. Sci. 304 (September 2019) 77–96, https://doi.org/10.4204/EPTCS.304.6, arXiv:1909.05972.

[6] Andi Bejleri, Elton Domnori, Malte Viering, Patrick Eugster, Mira Mezini, Comprehensive multiparty session types, Art Sci. Eng. Program. 3 (3) (February 2019) 6:1–6:59, https://doi.org/10.22152/programming-journal.org/2019/3/6.

[7] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Global progress in dynamically interleaved multiparty sessions, in: Franck van Breugel, Marsha Chechik (Eds.), CONCUR 2008 - Concurrency Theory, in: Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2008, pp. 418–433.

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

[9] Gérard Boudol, Asynchrony and the Pi-calculus, Research Report RR-1702, INRIA, 1992.

[10] Tevfik Bultan, Xiang Fu, Specification of realizable service conversations using collaboration diagrams, Serv. Oriented Comput. Appl. 2 (1) (April 2008) 27–39, https://doi.org/10.1007/s11761-008-0022-7.

[11] Luís Caires, Types and logic, concurrency and non-determinism, Technical Report MSR-TR-2014-104, in: Essays for the Luca Cardelli Fest, Microsoft Research, September 2014.
[12] Luís Caires, Jorge A. Pérez, Multiparty session types within a canonical binary theory, and beyond, in: Elvira Albert, Ivan Lanese (Eds.), Formal Techniques for Distributed Objects, Components, and Systems, in: Lecture Notes in Computer Science, Springer International Publishing, 2016, pp. 74–95.
[13] Luís Caires, Jorge A. Pérez, Linearity, control effects, and behavioral types, in: Hongseok Yang (Ed.), Programming Languages and Systems, in: Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2017, pp. 229–259.
[14] Luís Caires, Frank Pfenning, Session types as intuitionistic linear propositions, in: Paul Gastin, François Laroussinie (Eds.), CONCUR 2010 - Concurrency Theory, in: Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2010, pp. 222–236.
[15] Luís Caires, Frank Pfenning, Bernardo Toninho, Linear logic propositions as session types, Math. Struct. Comput. Sci. 26 (3) (2016) 367–423, https://doi.org/10.1017/S0960129514000218.
[16] Luís Caires, Hugo Torres Vieira, Conversation types, Theor. Comput. Sci. 411 (51) (December 2010) 4399–4440, https://doi.org/10.1016/j.tcs.2010.09.010.
[17] Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, Philip Wadler, Coherence generalises duality: a logical explanation of multiparty session types, in: Josée Desharnais, Radha Jagadeesan (Eds.), 27th International Conference on Concurrency Theory (CONCUR 2016), in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 59, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2016, pp. 33:1–33:15.
[18] Marco Carbone, Nobuko Yoshida, Kohei Honda, Asynchronous session types: exceptions and multiparty interactions, in: Marco Bernardo, Luca Padovani, Gianluigi Zavattaro (Eds.), Formal Methods for Web Services: 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2009, Bertinoro, Italy, June 1–6, 2009, Advanced Lectures, in: Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2009, pp. 187–212.
[19] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Luca Padovani, On global types and multi-party session, Log. Methods Comput. Sci. 8 (1) (March 2012), https://doi.org/10.2168/LMCS-8(1:24)2012.
[20] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, Paola Giannini, Ross Horne, Global types with internal delegation, Theor. Comput. Sci. 807 (February 2020) 128–153, https://doi.org/10.1016/j.tcs.2019.09.027.
[21] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, Nobuko Yoshida, Inference of global progress properties for dynamically interleaved multiparty sessions, in: Rocco De Nicola, Christine Julien (Eds.), Coordination Models and Languages, in: Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2013, pp. 45–59.
[22] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Luca Padovani, Global progress for dynamically interleaved multiparty sessions, Math. Struct. Comput. Sci. 26 (2) (February 2016) 238–302, https://doi.org/10.1017/S0960129514000188.
[23] 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, in: Lecture Notes in Computer Science, Springer International Publishing, 2018, pp. 91–109.
[24] Ornela Dardha, Elena Giachino, Davide Sangiorgi, Session types revisited, in: Danny De Schreye, Gerda Janssens, Andy King (Eds.), Principles and Practice of Declarative Programming, PPDP'12, Leuven, Belgium, September 19–21, 2012, ACM, 2012, pp. 139–150.
[25] Ornela Dardha, Jorge A. Pérez, Comparing deadlock-free session typed processes, Electron. Proc. Theor. Comput. Sci. 190 (August 2015) 1–15, https://doi.org/10.4204/EPTCS.190.1, arXiv:1508.06707.
[26] Ornela Dardha, Jorge A. Pérez, Comparing type systems for deadlock freedom, J. Log. Algebraic Methods Program. 124 (2022) 100717, https://doi.org/10.1016/j.jlamp.2021.100717.
[27] Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, Nobuko Yoshida, Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python, Form. Methods Syst. Des. 46 (3) (June 2015) 197–225, https://doi.org/10.1007/s10703-014-0218-8.
[28] Pierre-Malo Deniélou, Nobuko Yoshida, Multiparty compatibility in communicating automata: characterisation and synthesis of global session types, in: Fedor V. Fomin, Rusins Freivalds, Marta Kwiatkowska, David Peleg (Eds.), Automata, Languages, and Programming, in: Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2013, pp. 174–186.
[29] Henry DeYoung, Luís Caires, Frank Pfenning, Bernardo Toninho, Cut reduction in linear logic as asynchronous session-typed communication, in: Patrick Cégielski, Arnaud Durand (Eds.), Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 16, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2012, pp. 228–242.
[30] Ed Brinksma, LOTOS — a formal description technique based on the temporal ordering of observational behaviour, Technical Report ISO 8807:1989, International Organization for Standardization, February 1989.
[31] Adrian Francalanza, Jorge A. Pérez, César Sánchez, Runtime verification for decentralised and distributed systems, in: Ezio Bartocci, Yliès Falcone (Eds.), Lectures on Runtime Verification: Introductory and Advanced Topics, in: Lecture Notes in Computer Science, Springer International Publishing, Cham, 2018, pp. 176–210.
[32] Hubert Garavel, Radu Mateescu, Frédéric Lang, Wendelin Serwe, CADP 2006: a toolbox for the construction and analysis of distributed processes, in: Werner Damm, Holger Hermanns (Eds.), Computer Aided Verification, in: Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2007, pp. 158–163.
[33] Simon J. Gay, Peter Thiemann, Vasco T. Vasconcelos, Duality of session types: the final cut, Electron. Proc. Theor. Comput. Sci. 314 (April 2020) 23–33, https://doi.org/10.4204/EPTCS.314.3, arXiv:2004.01322.
[34] Blaise Genest, Anca Muscholl, Helmut Seidl, Marc Zeitoun, Infinite-state high-level MSCs: model-checking and realizability, J. Comput. Syst. Sci. 72 (4) (June 2006) 617–647, https://doi.org/10.1016/j.jcss.2005.09.007.
[35] Kohei Honda, Types for dyadic interaction, in: Eike Best (Ed.), CONCUR'93, in: Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 1993, pp. 509–523.
[36] Kohei Honda, Mario Tokoro, An object calculus for asynchronous communication, in: Pierre America (Ed.), ECOOP'91 European Conference on Object-Oriented Programming, in: Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 1991, pp. 133–147.
[37] Kohei Honda, Nobuko Yoshida, Marco Carbone, Multiparty asynchronous session types, in: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '08, Association for Computing Machinery, San Francisco, California, USA, January 2008, pp. 273–284.
[38] Kohei Honda, Nobuko Yoshida, Marco Carbone, Multiparty asynchronous session types, J. ACM 63 (1) (March 2016), https://doi.org/10.1145/2827695.
[39] Raymond Hu, Andi Bejleri, Nobuko Yoshida, Pierre-Malo Denielou, Parameterised multiparty session types, Log. Methods Comput. Sci. 8 (4) (October 2012), https://doi.org/10.2168/LMCS-8(4:6)2012.
[40] 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) (April 2016) 3:1–3:36, https://doi.org/10.1145/2873052.
[41] Keigo Imai, Rumyana Neykova, Nobuko Yoshida, Shoji Yuen, Multiparty session programming with global protocol combinators, in: Robert Hirschfeld, Tobias Pape (Eds.), 34th European Conference on Object-Oriented Programming (ECOOP 2020), in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 166, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2020, pp. 9:1–9:30.
[42] Limin Jia, Hannah Gommerstadt, Frank Pfenning, Monitors and blame assignment for higher-order session types, in: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16, ACM, New York, NY, USA, 2016, pp. 582–594.
[43] Naoki Kobayashi, Benjamin C. Pierce, David N. Turner, Linearity and the pi-calculus, ACM Trans. Program. Lang. Syst. 21 (5) (September 1999) 914–947, https://doi.org/10.1145/330249.330251.

[44] Rupak Majumdar, Nobuko Yoshida, Damien Zufferey, Multiparty motion coordination: from choreographies to robotics programs, in: Proceedings of the ACM on Programming Languages, 4(OOPSLA), November 2020, pp. 134:1–134:30.
[45] Rumyana Neykova, Laura Bocchi, Nobuko Yoshida, Timed runtime monitoring for multiparty conversations, Form. Asp. Comput. 29 (5) (September 2017) 877–910, https://doi.org/10.1007/s00165-017-0420-8.
[46] Luca Padovani, Vasco Thudichum Vasconcelos, Hugo Torres Vieira, Typing liveness in multiparty communicating systems, in: Eva Kühn, Rosario Pugliese (Eds.), Coordination Models and Languages, in: Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2014, pp. 147–162.
[47] C. Peltz, Web services orchestration and choreography, Computer 36 (10) (October 2003) 46–52, https://doi.org/10.1109/MC.2003.1236471.
[48] Benjamin C. Pierce, Types and Programming Languages, MIT Press, Cambridge, Massachusetts, 2002.
[49] G. Salaün, T. Bultan, N. Roohi, Realizability of choreographies using process algebra encodings, IEEE Trans. Serv. Comput. 5 (3) (Third Quarter 2012) 290–304, https://doi.org/10.1109/TSC.2011.9.
[50] Alceste Scalas, Ornela Dardha, Raymond Hu, Nobuko Yoshida, A linear decomposition of multiparty sessions for safe distributed programming, in: Peter Müller (Ed.), 31st European Conference on Object-Oriented Programming (ECOOP 2017), in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 74, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2017, pp. 24:1–24:31.
[51] Alceste Scalas, Nobuko Yoshida, Less is more: multiparty session types revisited, in: Proceedings of the ACM on Programming Languages, 3(POPL), January 2019, pp. 30:1–30:29. Revised, extended version at https://www.doc.ic.ac.uk/research/technicalreports/2018/DTRS18-6.pdf.
[52] Bernardo Toninho, Luis Caires, Frank Pfenning, Corecursion and non-divergence in session-typed processes, in: Matteo Maffei, Emilio Tuosto (Eds.), Trustworthy Global Computing, in: Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2014, pp. 159–175.
[53] Bernardo Toninho, Nobuko Yoshida, Interconnectability of session-based logical processes, ACM Trans. Program. Lang. Syst. 40 (4) (December 2018) 17, https://doi.org/10.1145/3242173.
[54] Bas van den Heuvel, Jorge A. Pérez, Deadlock freedom for asynchronous and cyclic process networks (extended version), arXiv:2111.13091 [cs], November 2021. A short version appears in the Proceedings of ICE'21: arXiv:2110.00146, arXiv:2111.13091.
[55] W.M.P. van der Aalst, Orchestration, in: Ling Liu, M. Tamer Özsu (Eds.), Encyclopedia of Database Systems, Springer US, Boston, MA, 2009, pp. 2004–2005.
[56] Vasco T. Vasconcelos, Fundamentals of session types, Inf. Comput. 217 (August 2012) 52–70, https://doi.org/10.1016/j.ic.2012.05.002.
[57] Philip Wadler, Propositions as sessions, in: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming, ICFP '12, ACM, New York, NY, USA, 2012, pp. 273–286.
[58] Nobuko Yoshida, Lorenzo Gheri, A very gentle introduction to multiparty session types, in: Dang Van Hung, Meenakshi D'Souza (Eds.), Distributed Computing and Internet Technology, in: Lecture Notes in Computer Science, Springer International Publishing, Cham, 2020, pp. 73–93.