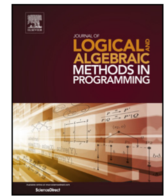
Journal of Logical and Algebraic Methods in Programming 102 (2019) 103–137

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp



# Session types and subtyping for orchestrated interactions

Franco Barbanera [a,*], Ugo de'Liguoro [b]

[a] Dipartimento di Matematica e Informatica, Università di Catania, Viale A. Doria 6, 95125 Catania, Italy
[b] Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy

ARTICLE INFO

*Article history:*
Received 8 December 2017
Accepted 14 May 2018
Available online 12 October 2018

*Keywords:*
Session types
Orchestration
Compliance
Explicit subtyping

ABSTRACT

In the setting of the $\pi$-calculus with binary sessions, we aim at relaxing the notion of duality of session types by the concept of *retractable compliance* developed in contract theory. This leads to extending session types with a new type operator of "speculative selection" including choices not necessarily offered by a compliant partner. We address the problem of selecting successful communicating branches by means of an operational semantics based on orchestrators, which has been shown to be equivalent to the retractable semantics of contracts, but clearly more feasible. A type system, sound with respect to such a semantics, is hence provided. The introduction of subtyping when interactions are orchestrated naturally leads to explicit subtyping, where coercions are functors on orchestrators. Besides, priority-governed selection policies (either at type- or process-level) are investigated in order to get rid of nondeterministic behaviours but those of the partner processes of the interactions.

© 2018 Elsevier Inc. All rights reserved.

## 1. Introduction

Service oriented software has fostered new scenarios in which software modules are supposed to meet each other, and to correctly cooperate, even if they do not descend from a coherent design, and are often produced by third parties. The central issue is adapting components on the basis of some abstract description of their behaviour, possibly by interposing interfaces to mediate their interaction. Since the discovery of a service that is compliant with the client's requirements might well happen at run time, it is essential that both checking for compatibility and building of interfaces are entirely automatic.

Contracts [1–4] and session types [5,6] are both intended as abstractions representing interaction protocols among concurrent processes. In both theories, interaction is modelled by message exchange along channels, abstractly represented by input/output actions indexed over channel names. Also the used formalisms all stem from CCS and its variants, possibly extended by some value passing mechanism. The resemblance is even tighter when considering "session contracts" [7,4], where only internal and external choices among contracts prefixed by, respectively, pairwise distinct output and input actions are allowed.

**Contracts versus session types.** In spite of similarities, these theories germinated from rather different concepts. In case of contracts, the input/output behaviour of a participant to a conversation is formalised as a whole by a term of an

* Corresponding author.
*E-mail addresses:* barba@dmi.unict.it (F. Barbanera), ugo.deliguoro@unito.it (U. de'Liguoro).

https://doi.org/10.1016/j.jlamp.2018.10.001
2352-2208/© 2018 Elsevier Inc. All rights reserved.

appropriate process algebra; contract theory then focuses on the "compliance" relation, holding when two or more protocols are such that, whenever there is an action by a participant that is expected to be performed, the symmetric, namely dual one is made available by some other participant. Restricting to the binary case, we say that a "server" protocol is compliant with a "client" one if all actions issued by the latter are matched by the respective co-actions issued by the former, possibly until the client reaches a successful state.

Session types are a type system for a dialect of Milner's $\pi$-calculus. Like with typed $\pi$-calculus, judgements associate to a process a "typing" that pairs channel names with the types of the values that can be transmitted through the channels; since $\pi$-terms are allowed to communicate channel names as well, channel types are among the types of exchanged values. Differently from ordinary $\pi$-calculus types, session types are regular trees of value types, that can be session types as well. In this way a single type can describe the flow of data through each channel that do not need to have all the same type; also input/output communication actions are distinguished by their types. When a "session" is opened by two processes in parallel, a new private channel is created – the session channel – that is shared by the processes; if the two "end-points", namely the respective occurrences of the session channel in the participant processes, are typed by *dual* types – roughly interchanging input and output types – then the interaction will be error free at run-time. Observe that not only the typing has to be checked against the process structure, which is not considered in contract theory, but also the very same process can issue several sessions at the same time, and session channels can be exchanged among processes. Therefore, even in the simpler setting of binary sessions, namely with channel names connecting two processes at a time, difficulties arise from the possible nesting of several sessions, and their ability to communicate across the boundary of a single session.

**Orchestrated compliance.** Compliance being a rather restrictive requirement, more liberal constraints have been proposed in the literature, among which are "orchestrated" compliance [8] and "retractable" compliance [9]. According to the orchestrated model, the interactions between a client and a server are mediated by a third process – the *orchestrator* – ruling interactions by allowing certain actions and co-actions only, possibly buffering a bounded number of messages on both sides. In the retractable model instead, actions are classified into irretractable (unaffectible) and retractable (affectible) ones. The concept is that, while irretractable actions by a participant have to be matched with their duals by the other compliant participant, retractable actions are just tried and possibly retracted, in case of communication failure, to issue some other action instead. Although these two models are different, it has been shown in [10] that, by restricting to certain orchestrators that allow just synchronous communications, contracts that are deemed compliant in both models are the same. Moreover, it is possible to provide an algorithm which synthesizes an orchestrator out of two retractable contracts if they are compliant, or fails otherwise (see [11] and Section 7 below).

**Orchestrated compliance instead of duality.** In this paper we address the issue of adapting the idea of retractable contracts to session types. More precisely we see session types as contracts, and propose to replace the otherwise restrictive notion of type duality by the relation of retractable or equivalently orchestrated compliance. To better illustrate the point, let us consider a process $P$ in parallel with a process $Q$. The former has to choose how to interact in a session with the latter by selecting one of several alternatives. Both $P$ and $Q$ are equipped with specification of their behaviours, so that it is known in advance that at least one alternative is actually successful, but not necessarily all of them are. Now there are two possible ways of guaranteeing $P$ to successfully complete the interaction with $Q$:

- either $P \mid Q$, representing the parallel composition of $P$ and $Q$, is run on a computational infrastructure that allows to roll back to some previous choice point $P' \mid Q'$ in case of synchronisation failure, and to try a different branch of the interaction;
- or, when checking the compliance of the specifications, it is statically computed which are the safe choices, if any, *before* running $P \mid Q$, so that they can be stored into a mediating process (the orchestrator).

The reason for preferring the second approach is clearly apparent, as it limits the backtracking to static type checking, while avoiding it at run-time. In fact, once types of opposite end-points have been recognized compliant up to retractability of certain choices (that are kept syntactically distinct from unretractable ones), and an orchestrator $\mathsf{f}$ has been synthesized, the very same orchestrator can serve as guidance in the interaction along the session channel. In particular, by putting the orchestrator $\mathsf{f}$ in parallel with the processes holding the end-points, it can be used to drive the proper choices at run time. If $k$ is the session channel, we write the resulting session by:

$$(\nu k)(\langle k\rangle \mathsf{f} \mid P \mid Q)$$

representing that the interaction over the private session channel $k$ is ruled by $\mathsf{f}$.

The orchestrators are abstractions that allow for many different implementations. One possibility would be running them as monitors on the communication infrastructure to inhibit certain actions on either side. A more effective realization, however, would be as wrappers, to be stored locally before running both server and client processes. To better appreciate the last solution consider the example in the paragraphs below, where a client is looking for a movie from a provider. Indeed the client is unlikely to buy or rent just one movie from the same provider; once the agreement has been found, by caching the code of the orchestrator we avoid both unfeasible modifications of the process code on either side, and time consuming renegotiations among the client and the server when repeated requests of the service may occur. Finally we

observe that an orchestrator may also serve to code an interaction policy, as illustrated below by the concept of speculative selection.

**Speculative selection.** In session-types formalism the type $\oplus\{l_1{:}S_1, \dots, l_n{:}S_n\}$ describes the protocol consisting in selecting one of the labels to be sent as output, say $l_i$, and continuing as specified in $S_i$; this is the consequence of an internal choice that is transparent to the other participant in the session, that is expected to be able to react to all $l_1, \dots, l_n$. The dual is the branching type $\&\{l_1{:}S'_1, \dots, l_n{:}S'_n\}$, expecting a label among $l_1, \dots, l_n$ as input to continue according to the respective continuation. To these we add a new type constructor written

$$\boxplus\{l_1{:}S_1, \dots, l_n{:}S_n\}$$

that we dub *speculative selection type*. The intended meaning of speculative selection is: try selecting labels among $l_1, \dots, l_n$ until an $l_i$ is found such that $l_i : S'_i$ is in the corresponding branching type, and $S_i$ and $S'_i$ are compliant. Observe that the speculative selection type has no dual, so that we cannot use the notion of duality in the system.

As a running example suppose that a *Client* is willing to establish a session with a movie-*Provider* and to behave on her channel end according to the following session type:

$$\mathsf{ClientSess} = \mathbf{!}[\mathtt{String}].\oplus\{\textsc{buy}{:}\oplus\{\textsc{uhd}{:}\mathsf{S}, \textsc{hd}{:}\mathsf{S}\},\ \textsc{rent}{:}\boxplus\{\textsc{uhd}{:}\mathsf{S}, \textsc{hd}{:}\mathsf{S}, \textsc{sd}{:}\mathsf{S}, \textsc{ld}{:}\mathsf{S}\}\}$$

where $\mathsf{S} = \mathbf{!}[\mathtt{String}].\&\{\textsc{ok}{:}\mathbf{?}[\mathtt{Url}], \textsc{no}{:}\mathsf{end}\}$.

The *Client* will send on the session channel a login information (an element of the ground type String) and then will internally decide whether she intends to BUY or to RENT a movie. In the former case, she will further decide whether to buy an ultra-high-definition (UHD) or a high-definition (HD) movie. In the latter case, instead, it is stated in the speculative selection type $\boxplus\{\textsc{uhd}{:}\mathsf{S}, \textsc{hd}{:}\mathsf{S}, \textsc{sd}{:}\mathsf{S}, \textsc{ld}{:}\mathsf{S}\}$ that she will proceed according to four possible failure-amenable choices: renting an ultra-high-definition (UHD), a high-definition (HD), a standard-definition (SD) or a low-definition (LD) movie. In all cases, she will proceed according to type S by sending the string with the title of the movie, and either receiving the URL from which the movie can be downloaded, if available (availability corresponds to the reception of OK), or ending the session if it is not (NO).

From the previous discussion, if the *Client* behaves on her end-point of the session channel according to the type ClientSess, she can safely interact with the *Provider* in case the latter behaves on the other end-point of the session channel according to the following session type:

$$\mathsf{ProvSess} = \mathbf{?}[\mathtt{String}].\&\{\textsc{buy}{:}\&\{\textsc{uhd}{:}\mathsf{S}', \textsc{hd}{:}\mathsf{S}'\},\ \textsc{rent}{:}\&\{\textsc{hd}{:}\mathbf{?}[\mathtt{Nat}].\mathsf{S}', \textsc{sd}{:}\mathsf{S}', \textsc{ld}{:}\mathsf{S}'\}\}$$

where $\mathsf{S}' = \mathbf{?}[\mathtt{String}].\oplus\{\textsc{ok}{:}\mathbf{!}[\mathtt{URL}], \textsc{no}{:}\mathsf{end}\}$.

Now this client/server interaction succeeds in case the client actually buys a movie. If the client intends to rent a movie, instead, the choice UHD will produce a synchronisation failure, since no ultra-high-definition movies are for rent on that server. Also the choice HD of a movie to be rented would produce a synchronisation failure, since then the server requires a membership code (as described by $\textsc{hd}{:}\mathbf{?}[\mathtt{Nat}].\mathsf{S}'$). But this is not all the story, as the remaining two possibilities lead to success, so that we insist that these two participants can agree at least in part. Indeed to prevent synchronisation failures the *Client* will be instructed at run-time by the orchestrator to select either the SD or the LD choice in case RENT had been previously selected. The orchestrator is computed when the session between the client and the server is tried and possibly opened, hence before it is started. An orchestrator for the example is:

$$\mathsf{f} = \bullet.((\textsc{buy}.(\textsc{uhd}.\mathsf{f}' + \textsc{hd}.\mathsf{f}')) + (\textsc{rent}.(\textsc{sd}.\mathsf{f}' \oplus \textsc{ld}.\mathsf{f}')))$$

where $\mathsf{f}' = \bullet.((\textsc{ok}.\bullet) + \textsc{no})$. This means that on the session channel between *Client* and *Provider*, the orchestrator f first enables an input/output interaction ($\bullet$) and then either ($+$) a BUY or a RENT branching. In case of RENT, the orchestrator internally decides ($\oplus$) to force either an SD or LD choice; this can be left open as both of them are "safe", namely do not lead to (not even future) synchronisation failures.

Observe that also the following two orchestrators can successfully drive the interaction between *Client* and *Provider*:

$$\mathsf{f}_1 = \bullet.((\textsc{buy}.(\textsc{uhd}.\mathsf{f}' + \textsc{hd}.\mathsf{f}')) + \textsc{rent}.\textsc{sd}.\mathsf{f}')$$

$$\mathsf{f}_2 = \bullet.((\textsc{buy}.(\textsc{uhd}.\mathsf{f}' + \textsc{hd}.\mathsf{f}')) + \textsc{rent}.\textsc{ld}.\mathsf{f}')$$

**Subtyping.** Subtype relations are developed with the goal of getting flexible type systems. As in [7], the subtyping relation depends on compliance which is not symmetric, therefore at least two (and possibly three, by considering the intersection of the two [12,13]) notions of subtyping naturally emerge.

How natural and feasible can be these notions of subtyping in the present *speculative* setting? In fact in this case the presence of orchestrators makes the definition and usage of a subtype relation not straightforward. The usual meaning of the subtyping statement $A \preccurlyeq B$ is that in case an interaction is safe by behaving according to $A$ on a channel-end, then it is so also by behaving according to $B$. Now, since in our setting interactions are mediated by orchestrators, when the interaction following $A$ is orchestrated by some f, it might be possible that the interaction following $B$ is orchestrated by

some $\mathsf{f}'$ different than $\mathsf{f}$. For the effectiveness of our system it is then important that $\mathsf{f}'$ depends uniformly on $\mathsf{f}$; by resorting to explicit subtyping we interpret $A \preccurlyeq B$ as the existence of a functor $F$ over orchestrators acting as some sort of coercion, such that if $\mathsf{f}$ orchestrates the interactions according to $A$, then $\mathsf{f}' = F(\mathsf{f})$ does the same with the interactions according to $B$: we write $F : A \preccurlyeq B$ when this is the case.

The use of functors in order to correctly deal with subtyping is essential not only for producing the orchestrator mediating two compatible processes, but also for dealing with the mechanism of delegation, since the flexibility provided by the use of subtyping affects the declared types of delegated channel-ends. Functors will hence be used to *adapt* the orchestrators of delegated sessions.

**Adding priorities.** In order to keep our calculus and its type system as general as possible, any safe orchestrator can be taken into account for a session interaction. In actual implementations, however, one could be interested in limiting the nondeterminism in a session exclusively to that exposed by the two partners. This involves considering *deterministic* orchestrators only, like $\mathsf{f}_1$ and $\mathsf{f}_2$ in the above *Client–Provider* example, that is with no $\oplus$ inside. For the semantics of the type $\boxplus\{\text{UHD}{:}\mathsf{S}, \text{HD}{:}\mathsf{S}, \text{SD}{:}\mathsf{S}, \text{LD}{:}\mathsf{S}\}$ the actual ordering of the labels is immaterial; therefore the actual choice stored in the orchestrator is up to the algorithm computing the orchestrator of a session. In practice, however, the choice of either $\mathsf{f}_1$ or $\mathsf{f}_2$ should not be randomly determined, rather it might reflect a particular policy representable at type level. A simple way is to provide a priority ordering among the failure-amenable choices, that can be expressed by the following modified syntax:

$$\boxplus\langle\!\langle \text{LD}{:}\mathsf{S}, \text{SD}{:}\mathsf{S}, \text{HD}{:}\mathsf{S}, \text{UHD}{:}\mathsf{S}\rangle\!\rangle$$

where $\langle\!\langle\ \rangle\!\rangle$ is now an ordered list. So the above type expresses that the option LD is the one liked best, but, in case of failure, SD is the second preferred choice, and so on. The priorities represented by the above speculative selection type force the orchestrator $\mathsf{f}_2$ to be synthesized.

We shall show that it is possible to synthesize the deterministic orchestrator which reflects the priorities described in the speculative selection types. Moreover, with an extra computational effort, the priority-ordering policy can be lowered down and described at process level. In particular, it is possible to synthesize orchestrators offering *all* the possible safe choices for speculative types. And then leaving to the programmer the task of specifying which one has to be chosen. This is done by representing in the process itself a priority list each time a speculative-selection operator is present. The nondeterministic behaviour of the orchestrators is hence resolved at run-time.

**Elimination of compliance-dependent deadlocks.** According to the compliance relation, in a session only the client's communicating actions are guaranteed to be matched by corresponding actions on the server's side. Therefore there might be some pending communications on the server side even in case the client has successfully completed. This fact produces particular deadlocks which do not show up in ordinary session-based calculi and type systems, as these are based on the notion of duality. We show how to get rid of such stuck states by adding suitable reduction rules. The type system can in fact be proved to be sound w.r.t. the new reductions, preventing typed process from reaching these peculiar stuck states.

In order to focus on the main concepts, in the present paper we do not treat recursion, that can be added in a fairly standard way.

**Structure of the paper.** In Section 2 we define types, orchestrators and the relation of orchestrated compliance. The syntax of the calculus and the type system are treated in Section 3. In Section 4 the operational semantics is defined and the subject reduction property is proved, obtaining error freeness of typed processes as a corollary. The particular deadlocks due to the use of the compliance relation instead of duality is dealt with in Section 5. In Section 6 we shall define subtyping with orchestrator functors as coercions and prove it to be a sound notion with respect to compliance-governed interactions. In Section 7 we propose how to use speculative types for actual programming, when nondeterminism is due only to the partners of the interactions. In particular we use orchestrators to implement priority-governed selection policies which can be described either at type- or process-level. Section 8 contains the conclusions and suggests possible extensions.

The present paper is a revised and extended version of [14], where proofs were mostly omitted, subtyping and orchestrator-functors as coercions were not present, and the use of priorities for speculative selection, both at type- and process-level, were only sketched.

## 2. Session types and orchestrated compliance

First we introduce session types following [5] but for recursion, omitted for the sake of simplicity, and for the new type $\boxplus\{l_i{:}S_i\}_{i\in I}$ for speculative selection, corresponding to retractable choice in [9].

**Definition 2.1** *(Types).* The set OST of Orchestrated Session Types is defined by the following grammar:

$$
\begin{array}{lllll}
G ::= \texttt{Nat} \mid \texttt{Bool} \mid \ldots & \text{ground types} & & \\
S ::= & \text{session types} & & \\
\quad \mid \text{end} & \text{terminated} & & \\
\quad \mid \mathbf{?}[G].S & \text{value input} & \mid \mathbf{!}[G].S & \text{value output} \\
\quad \mid \mathbf{?}[S_1^p].S_2 & \text{session input} & \mid \mathbf{!}[S_1^p].S_2 & \text{session output} \\
\quad \mid \bigoplus\{l_i{:}S_i\}_{i\in I} & \text{selection} & \mid \&\{l_i{:}S_i\}_{i\in I} & \text{branching} \\
\quad \mid \boxplus\{l_i{:}S_i\}_{i\in I} & \text{speculative selection} & & \\
T ::= G \mid S^p & & & \text{I/O types}
\end{array}
$$

where $p \in \{+, -, \pm\}$ and $I$ is a non-empty and finite set if indexes. Labels $l_i$'s belong to a countable set of labels $\mathcal{L}$ and are pairwise distinct in branching, selection and speculative-selection types.

The syntax of *orchestrators* provided by the following definition is similar to those present in [8] and [11]. Main differences with respect to [8] and [11] are that we do not have buffers, here unnecessary because we do not model asynchronous communications; also, for the sake of generality, orchestrators can introduce nondeterminism in orchestrated interactions (see Definition 4.2). Deterministic orchestrators will be introduced and discussed in Section 7.

**Definition 2.2** *(Orchestrators).* We define the set Orch of *orchestrators*, ranged over by f, g, ..., as the terms generated by the following grammar:

$$
\begin{array}{lll}
\mathsf{f}, \mathsf{g} ::= \mathbb{1} & \text{idle} \\
\quad \mid \bullet.\mathsf{f} & \text{I/O prefix} \\
\quad \mid l.\mathsf{f} & \text{selection prefix} \\
\quad \mid \sum_{i\in I} l_i.\mathsf{f}_i & \text{external choices} \\
\quad \mid \bigoplus_{i\in I} l_i.\mathsf{f}_i & \text{internal choices}
\end{array}
$$

where $l, l' \in \mathcal{L}$ and $I$ is a non-empty and finite set if indexes. Labels $l_i$' are assumed to be pairwise distinct both in internal and external choices.

We write $l_{i_1}.\mathsf{f}_{i_1} + \cdots + l_{i_n}.\mathsf{f}_{i_n}$ (resp. $l_{i_1}.\mathsf{f}_{i_1} \oplus \cdots \oplus l_{i_n}.\mathsf{f}_{i_n}$) for $\sum_{i\in I} l_i.\mathsf{f}_i$ (resp. $\oplus_{i\in I} l_i.\mathsf{f}_i$), where $I = \{i_1, \ldots, i_n\}$. If $I$ is a singleton then $\sum_{i\in I} l_i.\mathsf{f}_i$ (resp. $\oplus_{i\in I} l_i.\mathsf{f}_i$) is just a selection prefix. For the sake of readability, trailing $\mathbb{1}$'s will be often omitted in orchestrators.

**Definition 2.3** *(Orchestrated compliance).* The *orchestrated compliance* relation $\dashv \subseteq \mathsf{Orch} \times \mathsf{OST} \times \mathsf{OST}$ is defined by the formal system in Fig. 1, where $\mathsf{f} : S \dashv S'$ abbreviates $(\mathsf{f}, S, S') \in \dashv$.

If $\mathsf{f} : S \dashv S'$ we say that $S$ and $S'$ are **compliant by** f. We say that $S$ and $S'$ are **compliant**, written $S \dashv S'$, if $\mathsf{f} : S \dashv S'$ for some f.

When no ambiguity may arise, we simply write $\mathsf{f} : S \dashv S'$ when there exists a derivation for it in the system of Fig. 1.

Notice that, because of rules [Cmpl-⊞·&] and [Cmpl-&·⊞] (where any non-empty $H \subseteq I \cap J$ can be taken into account), we can have many orchestrators for the same pair of types.

**Proposition 2.4** *(Decidability). Given $S, S' \in \mathsf{OST}$, it is decidable whether $S \dashv S'$. Moreover if $S \dashv S'$ then an orchestrator* f *such that* $\mathsf{f} : S \dashv S'$ *is computable.*

**Proof.** By induction over the structure of $S, S'$. □

**Remark 2.5.** Both Definition 2.3 and Proposition 2.4 easily extend to the case of (contractive) recursive types and orchestrators (see [11], where orchestrated compliance is defined for "session contracts" instead of types).

**Remark 2.6.** In the theory of session contracts [12,4], compliance does correspond to the composition of duality and subtyping. Such a correspondence does transfer also to session types in a very general sense, as shown in [15]. In our setting,

$$[\text{CMPL-AX}]\quad \mathtt{1} : \mathsf{end} \dashv S$$

$$[\text{CMPL-?}\cdot\text{!VAL}]\ \frac{\mathsf{f} : S \dashv S'}{\bullet.\mathsf{f} : \mathbf{?}[T].S \dashv \mathbf{!}[T].S'} \qquad [\text{CMPL-!}\cdot\text{?VAL}]\ \frac{\mathsf{f} : S \dashv S'}{\bullet.\mathsf{f} : \mathbf{!}[T].S \dashv \mathbf{?}[T].S'}$$

$$[\text{CMPL-}\oplus\cdot\&]\ \frac{(\forall i \in I) \quad \mathsf{f}_i : S_i \dashv S'_i}{\sum_{i\in I} l_i.\mathsf{f}_i : \bigoplus\{l_i{:}S_i\}_{i\in I} \dashv \&\{l_j{:}S'_j\}_{j\in I\cup J}}$$

$$[\text{CMPL-}\&\cdot\oplus]\ \frac{(\forall i \in I) \quad \mathsf{f}_i : S_i \dashv S'_i}{\sum_{i\in I} l_i.\mathsf{f}_i : \&\{l_j{:}S_j\}_{j\in I\cup J} \dashv \bigoplus\{l_i{:}S'_i\}_{i\in I}}$$

$$[\text{CMPL-}\boxplus\cdot\&]\ \frac{\emptyset \neq H \subseteq I \cap J \qquad (\forall h \in H) \quad \mathsf{f}_h : S_h \dashv S'_h}{\oplus_{h\in H} l_h.\mathsf{f}_h : \boxplus\{l_i{:}S_i\}_{i\in I} \dashv \&\{l_j{:}S'_j\}_{j\in J}}$$

$$[\text{CMPL-}\&\cdot\boxplus]\ \frac{\emptyset \neq H \subseteq I \cap J \qquad (\forall h \in H) \quad \mathsf{f}_h : S_h \dashv S'_h}{\oplus_{h\in H} l_h.\mathsf{f}_h : \&\{l_j{:}S_j\}_{j\in J} \dashv \boxplus\{l_i{:}S'_i\}_{i\in I}}$$

**Fig. 1.** Orchestrated compliance relation.

however, even if a sound relation of subtyping can be defined for orchestrated session types (see Section 6), the above mentioned correspondence looks unrealistic since, as pointed out in [9], there exists no natural notion of duality[1] for retractable contracts.

**Example 2.7.** In case type $T$ in rules [CMPL-?·!VAL] and [CMPL-!·?VAL] from Fig. 1 is a session type, compliance is established among higher-order types that model delegation. We extend now the running example described in the Introduction to include the higher-order features of orchestrated session types. We assume that the *Client* of the movie-*Provider* has to pay for the buyed/rented movie. For paying an available movie, *Provider* throws to *Client* a session channel (typed by the session PAY) allowing payment by means of several possible cards. The new versions of ClntSess and ProvSess are now, respectively:

$$\mathsf{ClntSess} = \mathbf{!}[\mathtt{String}].\bigoplus\{\textsc{buy}{:}\bigoplus\{\textsc{uhd}{:}\mathsf{S}, \textsc{hd}{:}\mathsf{S}\},\ \textsc{rent}{:}\boxplus\{\textsc{uhd}{:}\mathsf{S}, \textsc{hd}{:}\mathsf{S}, \textsc{sd}{:}\mathsf{S}, \textsc{ld}{:}\mathsf{S}\}\}$$

where $\mathsf{S} = \mathbf{!}[\mathtt{String}].\bigoplus\{\textsc{ok}{:}\mathbf{?}[\mathsf{PAY}^-].\mathbf{?}[\mathtt{Url}], \textsc{no}{:}\mathsf{end}\}$

$$\mathsf{ProvSess} = \mathbf{?}[\mathtt{String}].\&\{\textsc{buy}{:}\&\{\textsc{uhd}{:}\mathsf{S}', \textsc{hd}{:}\mathsf{S}'\},\ \textsc{rent}{:}\&\{\textsc{hd}{:}\mathbf{?}[\mathtt{Nat}].\mathsf{S}', \textsc{sd}{:}\mathsf{S}', \textsc{ld}{:}\mathsf{S}'\}\}$$

where $\mathsf{S}' = \mathbf{?}[\mathtt{String}].\bigoplus\{\textsc{ok}{:}\mathbf{!}[\mathsf{PAY}^-].\mathbf{!}[\mathtt{Url}], \textsc{no}{:}\mathsf{end}\}$.

Here we assume for simplicity that the payment always succeeds and that *Client* is not cheating. The safe interaction between *Client* and *Provider* is guaranteed by $\mathsf{ClntSess} \dashv \mathsf{ProvSess}$. In fact, for instance, it can be checked that:

$$\mathsf{g} : \mathsf{ClntSess} \dashv \mathsf{ProvSess}$$

where

$$\mathsf{g} = \bullet.(\,(\textsc{buy}.(\textsc{uhd}.\mathsf{g}' + \textsc{hd}.\mathsf{g}')) + (\textsc{rent}.(\textsc{sd}.\mathsf{g}' \oplus \textsc{ld}.\mathsf{g}'))\,)$$

with $\mathsf{g}' = \bullet.(\,(\textsc{ok}.\bullet.\bullet) + \textsc{no}\,)$. Hence, in case of rental, the orchestrator g restricts the possible choices to the safe ones, namely SD and LD.

For paying, the *Provider* is assumed to establish a session with the *Bank*, acting as a client in this interaction. The *Provider*'s end of the corresponding session channel is typed by:

$$\mathsf{bankCustSess} = \mathbf{!}[\mathtt{Amount}].\mathsf{PAY}$$

where

[1] Duality for retractable session contracts can be immediately recovered by extending the formalism of [9] with speculative input choices, as done in [16]. Such extension, however, does not seem to have a clear session-type counterpart.

$$\mathsf{PAY} = \boxplus\{\text{DINERS}{:}\mathbf{!}[\mathtt{ccNumber}], \text{MCARD}{:}\mathbf{!}[\mathtt{ccNumber}], \text{VISA}{:}\mathbf{!}[\mathtt{ccNumber}]\}$$

Notice that PAY is precisely the type of the channel-end that the *Client* receives from the movie-*Provider* during the interaction described by ClntSess.

Hence, once the *Provider-Bank* session is established, the *Provider* sends the cost of the movie and then delegates the actual payment to the *Client*. The polarity '$-$' in the channel-send of ProvSess indicates that the delegated channel-end is sent by the applicant of the session. Therefore the receiver has to act as such.[2]

On the delegated channel the *Client* is allowed to pay by using either a DINERS, or a MASTERCARD or a VISA, but it is not guaranteed that all of them will be actually available.

We also assume that once the *Provider-Bank* session is established, the *Bank* behaves on its channel end according to the following type

$$\mathsf{BankSess} = \mathbf{?}[\mathtt{Amount}].\&\{\text{DISCOVER}{:}\mathsf{bS}, \text{MCARD}{:}\mathsf{bS}, \text{VISA}{:}\mathsf{bS}, \text{AEXPR}{:}\mathsf{bS}\}$$

where $\mathsf{bS} = \mathbf{?}[\mathtt{ccNumber}].\mathbf{!}[\mathtt{TransIDnum}]$.

Observe that the *Bank* can accept any of the DISCOVER, MCARD, VISA or AEXPR cards, but does not accept DINERS. Moreover, after receiving the credit-card number, the *Bank* does issue the identifier of the transaction (an element of the ground type TransIDnum), notwhithstanding it is not requested by the bankCustSess session type. Nonetheless the safety of the interaction with the *Bank* is guaranteed by the fact that bankCustSess $\dashv$ BankSess. In fact, for instance, it can be checked that

$$\mathsf{h} : \mathsf{bankCustSess} \dashv \mathsf{BankSess}$$

where $\mathsf{h} = \bullet.(\,(\text{MCARD}.\bullet) \oplus (\text{VISA}.\bullet)\,)$.

## 3. Calculus and type assignment

We assume to have a countable set $\mathcal{K}$ of *channel names*, ranged over by $k, k', \ldots$ In a session, we refer to the process performing the session-opening request as "the client", while the process which accepts the request is referred to as "the server".

We distinguish among user-defined and run-time processes. While the former represent the code of concurrent programs, the latter formalise the state of the system at run-time. Following [17,6], clients' and servers' channel-ends are identified by means of polarities: $-$ and $+$, respectively. Polarities are ranged over by $p, q, \ldots$ A channel-end of the form $k^p$ is called *polarized channel-end*, or simply *polarized channel*, for short. Moreover, we define $k^{\overline{+}} = k^-$ and $k^{\overline{-}} = k^+$.

**Definition 3.1** *(Processes).* The set of (user-defined) *pre-processes* is defined as the set of expressions generated by the following grammar:

| $P$ ::= | | | |
|---|---|---|---|
| $\mid\ \mathbf{0}$ | terminated | $\mid\ (\,P \mid P\,)$ | parallel |
| $\mid\ \mathtt{request}_S(k)P$ | session request | $\mid\ \mathtt{accept}_S(k)P$ | session accept |
| $\mid\ k![e].P$ | value send | $\mid\ k?(x).P$ | value receive |
| $\mid\ \mathtt{throw}\,k[k'].P$ | channel send | $\mid\ \mathtt{catch}\,k(k').P$ | channel receive |
| $\mid\ k\triangleleft l.P$ | selection | $\mid\ k \triangleright \{l_i{:}P_i\}_{i\in I}$ | branching |
| $\mid\ k\triangleleft [l_i{:}P_i]_{i\in I}$ | speculative sel. | | |

where $k, k' \in \mathcal{K}$, $I$ is non-empty and finite, and the labels in branching and speculative selection, all belonging to $\mathcal{L}$ (as the one in selection), are pairwise distinct.

The channel name $k$ is bound in $\mathtt{request}_S(k)P$, $\mathtt{accept}_S(k)P$, but free in $\mathtt{catch}\,k(k').P$ where $k'$ is bound. Similarly the expression variable $x$ is bound in $k?(x).P$. All other occurrences of $k, k'$ and $x$ are free.

*User-defined processes* are closed pre-processes, namely pre-processes without free occurrences of channel names nor of expression variables.

Let $P$ be any user defined pre-process. *Run-time processes* are defined by the grammar

$$R ::= P \mid (\nu k)(\langle k\rangle \mathsf{f} \mid R) \mid (R \mid R')$$

where channels can be *polarized channels*. In $(\nu k)(\langle k\rangle \mathsf{f} \mid R)$ the operator $(\nu k)$ is a binder of $k^p$, $k^{\overline{p}}$ and $k$.

[2] The polarities of two corresponding channel-send and channel-receive are always the same, since they both refer to the same channel's end; in particular, in the present example the polarity is '$-$' also in the channel-receive of ClntSess.

In session type systems [5,6] a port name $a$ is added in request $a(k)P$ and accept $a(k)Q$, then $a$ is typed by a pair $\langle S, \overline{S}\rangle$ of dual types, so that $a$ (in fact $k$) has type $S$ in request $a(k)P$ and $\overline{S}$ in accept $a(k)Q$. Here we do not have any plausible notion of *dual type*, on the contrary we are relaxing this constraint by means of compliance. Still it does make sense to have port names as the formal counterpart of specific addresses in our context; however there is no reason to pair processes by their names, whereas just compliance of the respective types matters. Therefore in our theoretical calculus (not a full fledged language) we ignore port names and label processes of the form $\texttt{request}_S(k)P$ and $\texttt{accept}_{S'}(k)P$ by their types only.

Most process actions are similar to those of the calculus in [5], but for the new *speculative selection*. A process $k \lhd [l_i{:}P_i]_{i\in I}$ is able to send on channel $k$ any of the labels $l_i$'s and to proceed afterwards as $P_i$. The process is aware that some of the synchronizations on the $l_i$'s could lead to a failure and an orchestrator is hence expected to drive the choice.

The processes *Provider*, *Client* and *Bank* of the running example can be described as follows (where also conditional processes are taken into account, which can be treated in a fairly standard way as in [5]). For the sake of conciseness, only the parts concerning rentals are described.

**Example 3.2.** Let **b** be a boolean expression representing the decision of whether buying or renting a movie.

$$
\begin{aligned}
&\mathit{Client} = \texttt{request}_{\mathsf{ClntSess}}(k)k![\text{loginfo}].\texttt{if}\ \mathbf{b}\ \texttt{then}\ k \lhd \textsc{buy}.C_{\textsc{b}}\ \texttt{else}\ k \lhd \textsc{rent}.C_{\textsc{r}}\\
&\quad \text{where } C_{\textsc{r}} = k \lhd [\textsc{uhd}.C', \textsc{hd}.C', \textsc{sd}.C', \textsc{ld}.C']\\
&\qquad \text{with } C' = k![''\mathit{zootropolis}''].k \rhd \{\textsc{ok}{:}\, C'_{\textsc{ok}}, \textsc{no}{:}\mathbf{0}\}\\
&\qquad\quad \text{where } C'_{\textsc{ok}} = \texttt{catch}\, k(k').k' \lhd [\textsc{Diners}{:}Q, \textsc{MCard}{:}Q, \textsc{Visa}{:}Q]\\
&\qquad\qquad \text{where } Q = k'![1234].k?(\mathit{url}).\mathit{Watch}(\mathit{url})\\
&\mathit{Provider} = \texttt{accept}_{\mathsf{ProvSess}}(k)k?(x).k \rhd \{\textsc{buy}{:}P_{\textsc{b}}, \textsc{rent}{:}P_{\textsc{r}}\}\\
&\quad \text{where } P_{\textsc{r}} = k \rhd \{\textsc{uhd}.P', \textsc{hd}.P', \textsc{sd}.P', \textsc{ld}.P'\}\\
&\qquad \text{with } P' = k?(y).\texttt{if}\ \text{available}(y)\ \texttt{then}\ k \lhd \textsc{ok}.P'_{\textsc{ok}}\ \texttt{else}\ k \lhd \textsc{no}\\
&\qquad\quad \text{where } P'_{\textsc{ok}} = \texttt{request}_{\mathsf{bnkCustSess}}(k')k'![\text{amount(y)}].\texttt{throw}\, k[k'].k![\text{url(y)}]\\
&\mathit{Bank} = \texttt{accept}_{\mathsf{BankSess}}(k')k'?(x).k' \rhd \{\textsc{Discover}{:}B, \textsc{MCard}{:}B, \textsc{Visa}{:}B, \textsc{AExpr}{:}B\}\\
&\quad \text{where } B = k'?(cc).k'![\text{IDtrans}(x, cc)]
\end{aligned}
$$

### 3.1. The type system

The following type system is like the "more liberal" system in [6] for what concerns the use of polarized channel-ends. However, whereas in [6] the compatibility of channel-ends is formalized by the relation of duality, we use instead the relation of orchestrated compliance.

For the sake of conciseness, in the following we shall ambiguously use the notation $P$ for both user-defined and run-time processes.

There are two kinds of judgements:

$$\Gamma \vdash e : G$$

where $e$ is a value-expression and $G$ is a ground type;

$$\Gamma \Vdash P \rhd \Delta$$

The *context* $\Gamma$ is a finite set of typings $x : G$ for expression variables; the *channel-typing* (henceforth just *typing*) $\Delta$ is a finite set of typings $k^p : S$, where $k^p$ is a polarised channel name, and $S$ a session type.

By $dom(\Gamma)$ and $dom(\Delta)$ we mean the set of variables or channel names that are typed, respectively, in $\Gamma$ and $\Delta$. Variables and (polarized) channel names are pairwise distinct both in $\Gamma$ and $\Delta$ as usual; note that $k^+ \neq k^-$.

**Definition 3.3** *(Type system).* The rules of the type system are in Fig. 2. In rule [Inact-T] a typing $\Delta$ is *completed* if for any $k^p \in dom(\Delta)$ we have $k^p : \mathsf{end} \in \Delta$. In rule [Conc-T] the typing $\Delta \cdot \Delta'$ is the union of the typings $\Delta$ and $\Delta'$ provided that $dom(\Delta) \cap dom(\Delta') = \emptyset$, it is undefined otherwise; in the latter case the rule does not apply.

The rules of Fig. 2 are similar to those of [6], but for the following ones.

In rules [Acc-T], [Req-T] port names are not considered; consequently contexts $\Gamma$ just contain value-variables and we do not have restrictions over port names. In [6] rule [CRes-T] is

$$\text{[CRes-T]}\ \frac{\Gamma \Vdash P \rhd \Delta \cdot k^- : S \cdot k^+ : \overline{S}}{\Gamma \Vdash (\nu k)P \rhd \Delta}$$

[Inact-T]
$$\frac{\Delta \text{ completed}}{\Gamma \Vdash \mathbf{0} \rhd \Delta}$$

[Conc-T]
$$\frac{\Gamma \Vdash P \rhd \Delta \quad \Gamma \Vdash Q \rhd \Delta'}{\Gamma \Vdash P \mid Q \rhd \Delta \cdot \Delta'}$$

[Acc-T]
$$\frac{\Gamma \Vdash P\{k^+/k\} \rhd \Delta \cdot k^+ : S}{\Gamma \Vdash \mathtt{accept}_S(k)P \rhd \Delta}$$

[Req-T]
$$\frac{\Gamma \Vdash P\{k^-/k\} \rhd \Delta \cdot k^- : S}{\Gamma \Vdash \mathtt{request}_S(k)P \rhd \Delta}$$

[Rec-T]
$$\frac{\Gamma, x : G \Vdash P \rhd \Delta \cdot k^p : S}{\Gamma \Vdash k^p?(x).P \rhd \Delta \cdot k^p : \mathbf{?}[G].S}$$

[Send-T]
$$\frac{\Gamma \vdash e : G \quad \Gamma \Vdash P \rhd \Delta \cdot k^p : S}{\Gamma \Vdash k^p![e].P \rhd \Delta \cdot k^p : \mathbf{!}[G].S}$$

[Cat-T]
$$\frac{\Gamma \Vdash P\{k'^q/k'\} \rhd \Delta \cdot k^p : S_2 \cdot k'^q : S_1}{\Gamma \Vdash \mathtt{catch}\, k^p(k').P \rhd \Delta \cdot k^p : \mathbf{?}[S_1^q].S_2}$$

[Thr-T]
$$\frac{\Gamma \Vdash P \rhd \Delta \cdot k^p : S_2}{\Gamma \Vdash \mathtt{throw}\, k^p[k'^q].P \rhd \Delta \cdot k^p : \mathbf{!}[S_1^q].S_2 \cdot k'^q : S_1}$$

[Br-T]
$$\frac{\forall i \in I \supseteq J \;\; \Gamma \Vdash P_i \rhd \Delta \cdot k^p : S_i}{\Gamma \Vdash k^p \rhd \{l_i : P_i\}_{i \in I} \rhd \Delta \cdot k^p : \&\{l_j{:}S_j\}_{j \in J}}$$

[Sel-T]
$$\frac{\Gamma \Vdash P \rhd \Delta \cdot k^p : S_j \quad j \in I}{\Gamma \Vdash k^p \lhd l_j.P \rhd \Delta \cdot k^p : \bigoplus\{l_i{:}S_i\}_{i \in I}}$$

[SSel-T]
$$\frac{\forall i \in I \;\; \Gamma \Vdash P_i \rhd \Delta \cdot k^p : S_i}{\Gamma \Vdash k^p \lhd [l_i : P_i]_{i \in I} \rhd \Delta \cdot k^p : \boxplus\{l_i{:}S_i\}_{i \in I}}$$

[CRes-T]
$$\frac{\Gamma \Vdash P \rhd \Delta \cdot k^- : S_1 \cdot k^+ : S_2 \qquad \mathsf{f} : S_1 \dashv S_2}{\Gamma \Vdash (\nu k)(\langle k \rangle \mathsf{f} \mid P) \rhd \Delta}$$

[CRes'-T]
$$\frac{\Gamma \Vdash P \rhd \Delta \qquad k^+, k^- \notin dom(\Delta)}{\Gamma \Vdash (\nu k)P \rhd \Delta}$$

**Fig. 2.** The type system.

where $\overline{S}$ is the dual of $S$. In the premise of our rule, we have instead the typing $\Delta \cdot k^- : S \cdot k^+ : S'$, with the side condition $\mathsf{f} : S \dashv S'$ yielding the orchestrator appearing in the process in the conclusion.

It is not difficult to check the parallel composition of the processes of our running example is a typable user-defined process, in particular, it is possible to derive

$$\emptyset \Vdash \mathit{Provider} \mid \mathit{Client} \mid \mathit{Bank} \rhd \emptyset$$

## 4. Operational semantics and error freeness

Because of the presence of orchestrators in session interactions, the notion of structural congruence has to be handled with some extra care than in usual calculi with session-types.

**Definition 4.1** *(Structural congruence).*
*Structural congruence* $\equiv$ is the least congruence over run-time processes such that:

1. $R \equiv R'$ if $R'$ is obtained from $R$ by alphabetical change of bound channel names, avoiding name clashes,
2. $(\nu k)(\; \langle k \rangle \mathsf{f} \;\mid\; Q \;\mid (\nu k')(\langle k' \rangle \mathsf{g} \mid Q' \mid R)) \equiv \;\; (\nu k')(\; \langle k' \rangle \mathsf{g} \;\mid\; (\nu k)(\langle k \rangle \mathsf{f} \mid Q \mid Q'\,) \;\mid\; R) \qquad$ *if* $k \notin \textsc{fc}(R), k' \notin \textsc{fc}(Q)$,
3. if $X, Y, Z$ are either user-defined or run-time processes that are not a named orchestrator, then: $(X \mid Y) \equiv (Y \mid X)$ and $(X \mid (Y \mid Z)) \equiv ((X \mid Y) \mid Z)$.

**Definition 4.2** *(Operational semantics).* The operational semantics of processes is described by the reduction relation defined by the rules listed in Fig. 3.

[LINK]
$$\mathtt{request}_S(k)P \mid \mathtt{accept}_{S'}(k)Q \longrightarrow (\nu k)(\,\langle k\rangle \mathsf{f} \mid P\{k^-/k\} \mid Q\{k^+/k\}\,) \qquad \text{if } \mathsf{f}: S \dashv S'$$

[ORCHCOMM]
$$(\nu k)(\,\langle k\rangle \bullet .\mathsf{f} \mid k^p![e].P \mid k^{\overline{p}}?(x).Q\,) \longrightarrow (\nu k)(\,\langle k\rangle \mathsf{f} \mid P \mid Q\{v/x\}\,) \qquad \text{if } e \downarrow v$$

[ORCHDELEG]
$$(\nu k)(\langle k\rangle \bullet .\mathsf{f} \mid \mathtt{throw}\, k^p[k'^q].P \mid \mathtt{catch}\, k^{\overline{p}}(k').Q\,) \longrightarrow (\nu k)(\langle k\rangle \mathsf{f} \mid P \mid Q\{k'^q/k'\})$$

[ORCHSEL]
$$(\nu k)(\,\langle k\rangle \textstyle\sum_{h\in H} l_h.\mathsf{f}_h \mid k^p \lhd l_c.P \mid k^{\overline{p}} \rhd \{l_i{:}Q_i\}_{i\in I}\,) \longrightarrow (\nu k)(\,\langle k\rangle \mathsf{f}_c \mid P \mid Q_c\,) \qquad \text{if } c \in H \cap I$$

[ORCHSSEL]
$$(\nu k)(\langle k\rangle \oplus_{h\in H} l_h.\mathsf{f}_h \mid k^p \lhd [l_j : P_j]_{j\in J} \mid k^{\overline{p}} \rhd \{l_i{:}Q_i\}_{i\in I}) \longrightarrow (\nu k)(\langle k\rangle \mathsf{f}_c \mid P_c \mid Q_c) \qquad \text{if } c \in H \cap I \cap J$$

[PAR]
$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$$

[SCOP]
$$\frac{P \longrightarrow P'}{(\nu k)P \longrightarrow (\nu k)P'}$$

[STR]
$$\frac{Q \equiv P \longrightarrow P' \equiv Q'}{Q \longrightarrow Q'}$$

**Fig. 3.** Operational semantics.

This operational semantics extends that in [5] by taking into account the new operator and the need for interactions of being orchestrated.

[LINK] If the session type $S$ is compliant with $S'$ by means of the orchestrator $\mathsf{f}$, the session-opening request $\mathtt{request}_S(k)P$ can be accepted by $\mathtt{accept}_{S'}(k)Q$. A new channel is created for the opened session. The '$-$' end is owned by the process who requested the opening (the client) whereas the '$+$' end is owned by the other one (the server). To connect the orchestrator $\mathsf{f}$ to the opened session, $\mathsf{f}$ is labelled with the channel name $k$. This forces its orchestration actions to act only on synchronisations over the channel $k$.

[ORCHCOMM], [ORCHDELEG], [ORCHSEL] In these rules the orchestrator enables the communication of a value, the communication of a channel, and the selection of a label, respectively. In rule [ORCHCOMM] the side condition $e \downarrow v$ reads: expression $e$ evaluates to the value $v$.

[ORCHSSEL] In presence of a speculative selection, i.e. a number of choices possibly leading to synchronisation failures, the role of the orchestrator is to suggest one among the safe choices. Notice that, in case the cardinality of $H \cap I \cap J$ is strictly greater than one, this rule is nondeterministic. In actual implementations it is reasonable to expect the orchestrator not to add nondeterminism to the system. We show in Section 7 how this can be obtained by interpreting $[l_j : P_j]_{j\in J}$ in $k^p \lhd [l_j : P_j]_{j\in J}$ as a priority list and how is it possible to synthesize, out of $S$ and $S'$ in [LINK], the orchestrator that suggests the safe choice possessing the highest priority, if any.

[PAR], [SCOP], [STR] These rules are standard.

**Example 4.3.** We can now see in Fig. 4 the evolution of the system of our running example, represented by the user-defined process

$$\mathit{Client} \mid \mathit{Provider} \mid \mathit{Bank}$$

where *Client*, *Provider* and *Bank* are as described in Example 3.2. We assume that the client decides to rent an available movie.

In order to show the flexibility of our approach, we modify now our running example by assuming two providers to be available. They behave in different ways (i.e. have different codes) but both their types are compatible with the client's type. This implies that the client may initiate a session with either of the two servers; different orchestrators can hence be generated, depending on the pair that actually synchronises.

**Example 4.4.** We extend the *Client* | *Provider* | *Bank* system with another movie provider: *ProviderII*. This second provider allows also for rentals of X-rated movies in HD. In that case, the provider performs an age-check based on the login information; no membership code is requested by *ProviderII* in case of a rental of an HD movie, so no synchronisation failure would occur in that case. The movies for rental of *ProviderII* are HD movies (which can be X-rated), SD and LD movies (which are not X-rated). Moreover, in case a movie cannot be rented, a string is sent back specifying the reason.

$$
\begin{array}{ll}
& \mathit{Client} \mid \mathit{Provider} \mid \mathit{Bank} \\[4pt]
\xrightarrow{[\text{LINK}]} & (\nu k)\big(\; k![\text{loginfo}].\texttt{if}\ \mathbf{b}\ \texttt{then}\ k\triangleleft\text{BUY}.C_{\text{B}}\ \texttt{else}\ k\triangleleft\text{RENT}.C_{\text{R}} \\
& \qquad \mid k?(x).k\triangleright\{\text{BUY}{:}P_{\text{B}},\ \text{RENT}{:}P_{\text{R}}\} \\
& \qquad \mid \langle k\rangle \mathsf{g} \;\big) \\
& \ \mid\ \mathit{Bank} \\[4pt]
\xrightarrow{[\text{ORCHCOMM}][\text{IF}][\text{ORCHSEL}]} & (\nu k)\big(C_{\text{R}} \mid P_{\text{R}} \mid \langle k\rangle \text{SD}.\mathsf{g}' \oplus \text{LD}.\mathsf{g}'\big) \mid \mathit{Bank} \\[4pt]
\xrightarrow{[\text{ORCHSSEL}]} & (\nu k)\big(C' \mid P' \mid \langle k\rangle \bullet .(\,(\text{OK}.\bullet.\bullet) + \text{NO}\,)\big) \mid \mathit{Bank} \\[4pt]
\xrightarrow{[\text{ORCHCOMM}][\text{IF}][\text{ORCHSEL}]} & (\nu k)\big(C'_{\text{OK}} \mid P'_{\text{OK}} \mid \langle k\rangle \bullet.\bullet\,\big) \mid \mathit{Bank} \\[4pt]
\xrightarrow{[\text{LINK}]} & (\nu k)(\nu k')\big(C'_{\text{OK}} \mid k'![\text{amount}(\text{“zootropolis”})].\texttt{throw}\,k[k'].k![\text{url}(\text{“zootropolis”})] \\
& \qquad \mid \langle k\rangle \bullet.\bullet \ \mid\ \langle k'\rangle \bullet.(\text{MCARD}.\bullet \oplus \text{VISA}.\bullet) \\
& \qquad \mid k'?(x).k'\triangleright\{\text{DISCOVER}{:}B,\ \text{MCARD}{:}B,\ \text{VISA}{:}B,\ \text{AEXPR}{:}B\}\big) \\[4pt]
\xrightarrow{[\text{ORCHCOMM}]} & (\nu k)(\nu k')\big(C'_{\text{OK}} \mid \texttt{throw}\,k[k'].k![\text{url}(\text{“zootropolis”})] \\
& \qquad \mid \langle k\rangle \bullet.\bullet \ \mid\ \langle k'\rangle(\text{MCARD}.\bullet \oplus \text{VISA}.\bullet) \\
& \qquad \mid k'\triangleright\{\text{DISCOVER}{:}B',\ \text{MCARD}{:}B',\ \text{VISA}{:}B',\ \text{AEXPR}{:}B'\}\big) \\
& \text{where } B' \ =\ k'?(cc).k'![\text{IDtrans}(\text{amount}(\text{“zootropolis”}), cc)] \\[4pt]
\xrightarrow{[\text{ORCHDELEG}]} & (\nu k)(\nu k')\big(\; k'\triangleleft[\text{DINERS}{:}Q,\ \text{MCARD}{:}Q,\ \text{VISA}{:}Q\,] \ \mid\ k![\text{url}(\text{“zootropolis”})] \\
& \qquad \mid \langle k\rangle\bullet \ \mid\ \langle k'\rangle(\text{MCARD}.\bullet \oplus \text{VISA}.\bullet) \\
& \qquad \mid k'\triangleright\{\text{DISCOVER}{:}B',\ \text{MCARD}{:}B',\ \text{VISA}{:}B',\ \text{AEXPR}{:}B'\}\big) \\
\xrightarrow{[\text{ORCHSSEL}]} & (\nu k)(\nu k')\big(\; k'![1234].k?(url).\mathit{Watch}(url) \ \mid\ k![\text{url}(\text{“zootropolis”})] \\
& \qquad \mid \langle k\rangle\bullet \ \mid\ \langle k'\rangle\bullet \ \mid\ k'?(cc).k'![\text{IDtrans}(\text{amount}(\text{“zootropolis”}), cc)] \;\big) \\
\xrightarrow{[\text{ORCHCOMM}]} & (\nu k)(\nu k')\big(\; k?(url).\mathit{Watch}(url) \ \mid\ k![\text{url}(\text{“zootropolis”})] \\
& \qquad \mid \langle k\rangle\bullet \ \mid\ \langle k'\rangle 1 \ \mid\ k'![\text{IDtrans}(\text{amount}(\text{“zootropolis”}), 1234)] \;\big) \\[4pt]
\xrightarrow{[\text{ORCHCOMM}]} & (\nu k)(\nu k')\big(\; \mathit{Watch}(\text{url}(\text{“zootropolis”})) \ \mid\ \mathbf{0} \\
& \qquad \mid \langle k\rangle 1 \ \mid\ \langle k'\rangle 1 \ \mid\ k'![\text{IDtrans}(\text{amount}(\text{“zootropolis”}), 1234)] \;\big)
\end{array}
$$

**Fig. 4.** Reductions example.

As in Example 3.2, we describe only the code of the “rental branch”.

$$
\begin{array}{rcl}
\mathit{ProviderII} & = & \texttt{accept}_{\mathsf{ProvSessII}}(k)k?(x).k\triangleright\{\text{BUY}{:}P'_{\text{B}},\ \text{RENT}{:}P'_{\text{R}}\} \\
\text{where } P'_{\text{R}} & = & k\triangleright\{\text{HD}.P'',\ \text{SD}.P''',\ \text{LD}.P'''\} \\
\text{with } P'' & = & k?(y).\texttt{if}\ \text{available}(y) \\
& & \quad \texttt{then}\ \texttt{if}\ \text{X-rated}(y) \\
& & \qquad \texttt{then}\ \texttt{if}\ \text{adult}(x) \\
& & \qquad\quad \texttt{then}\ k\triangleleft\text{OK}.P'_{\text{OK}} \\
& & \qquad\quad \texttt{else}\ k\triangleleft\text{NO}.k![\text{“X-rated movie”}] \\
& & \qquad \texttt{else}\ k\triangleleft\text{OK}.P'_{\text{OK}} \\
& & \quad \texttt{else}\ k\triangleleft\text{NO}.k![\text{“movie not available”}] \\
\text{and } P''' & = & k?(y).\texttt{if}\ \text{available}(y) \\
& & \quad \texttt{then}\ k\triangleleft\text{OK}.P'_{\text{OK}} \\
& & \quad \texttt{else}\ k\triangleleft\text{NO}.k![\text{“movie not available”}]
\end{array}
$$

where $P'_{\text{OK}}$ is as in Example 3.2

and where

$$
\begin{array}{l}
\mathsf{ProvSessII} \ = \mathbf{?}[\texttt{String}].\&\{\text{BUY}{:}\&\{\text{UHD}{:}\mathsf{S}'',\ \text{HD}{:}\mathsf{S}''\},\ \ \text{RENT}{:}\&\{\text{HD}{:}\mathsf{S}'',\ \text{SD}{:}\mathsf{S}'',\ \text{LD}{:}\mathsf{S}''\}\} \\[4pt]
\qquad \text{with } \mathsf{S}'' \ = \mathbf{?}[\texttt{String}].\bigoplus\{\text{OK}{:}\mathbf{!}[\mathsf{PAY}^{-}].\mathbf{!}[\texttt{Url}],\ \text{NO}{:}\mathbf{!}[\texttt{String}]\}
\end{array}
$$

In the new system

$$\mathit{Client} \mid \mathit{Provider} \mid \mathit{ProviderII} \mid \mathit{Bank}$$

*Client* can connect to either *Provider* or *ProviderII*. In the first case the system evolves like in Fig. 4, and the interaction is mediated by the orchestrator g described in Example 2.7. In the second case the interaction is mediated instead by a different orchestrator allowing also the rental of HD movies, namely

$$\bullet.(\,(\text{BUY}.(\text{UHD}.g' + \text{HD}.g')) + (\text{RENT}.(\text{HD}.g' \oplus \text{SD}.g' \oplus \text{LD}.g'\,)$$

where $g'$ is as described in Example 2.7.

The type system guarantees type-checked processes to be free from (a version of) the standard synchronisation errors of session-type-based calculi, which now involve also orchestrators and that we dub *orchestrated synchronisation errors*. It also prevents (a class of) errors due to the absence of orchestration action that we dub *vacuous orchestration errors*.

**Example 4.5.** Let us see some examples of errors that cannot actually occur in typeable processes (see Lemma 4.7 below).

$(\nu k)(k^+![e].R \mid k^-![e'].R' \mid \langle k\rangle f)$ This process is stuck. It cannot be typed since the types for $k^+$ and $k^-$ should have, respectively, the form $\mathbf{!}[G].S_1$ and $\mathbf{!}[G'].S_2$. Rule [CRes-T] cannot be applied to type the whole process, since, by Definition 2.3, these types are not compliant (even in case $G = G'$).

$(\nu k)(k^+\triangleleft l.R \mid k^+ \triangleright \{l_j{:}R_j\}_{j\in J} \mid \langle k\rangle \sum_{i\in I} l_i.f_i)$ This stuck process cannot be typed since rule [CRes-T] requires the compliant types to be assigned to polarized channels with different polarities.

$(\nu k)(k^+![e].R \mid k^-?(x).R' \mid \langle k\rangle \sum_{i\in I} l_i.f_i)$ This process is stuck since the orchestrator does not enable the input/output synchronisation. It cannot be typed either. In fact the types for $k^+$ and $k^-$ should have, respectively, the form $\mathbf{!}[G].S_1$ and $\mathbf{?}[G].S_2$, and rule [CRes-T] cannot be applied since, by Definition 2.3, $\sum_{i\in I} l_i.f_i : \mathbf{!}[G].S_1 \dashv \mathbf{?}[G].S_2$ is impossible.

$(\nu k)(k^-![e].R \mid R' \mid \langle k\rangle 1)$ Since any reduction is necessarily driven by an orchestrating action, there is no possibility for the process $k^-![e].R$ to progress. Unlike the previous examples, this sort of deadlock depends exclusively on the lack of orchestration actions. It cannot be typed, since the type for $k^-$ should have the form $\mathbf{!}[G].S'$ and hence rule [CRes-T] should be used with its side condition having the form $1 : \mathbf{!}[G].S' \dashv S$, which is impossible by Definition 2.3.

**Definition 4.6** *(Errors).*

i) A $k^p$*-process* is a run-time process term whose first action involves the channel $k$, namely a process having one of the following forms:

$$k^p![e].P, \quad k^p?(x).P, \quad \texttt{throw}\,k^p[k'^q].P, \quad \texttt{catch}\,k^p(k'^q).P,$$
$$k^p\triangleleft l.P, \quad k^p \triangleright \{l_i{:}P_i\}_{i\in I}, \quad k^p \triangleleft [l_i{:}P_i]_{i\in I};$$

ii) A *potential k-redex* is a process term formed by the $(\nu k)$-restricted parallel composition of one $k^p$-process, one $k^q$-process, for some $p$ and $q$, and one $k$-named orchestrator;

iii) A process $R$ is an *orchestrated synchronisation error* (orch-synch error, for short) if it contains a potential *k-redex* which does not reduce;

iv) A process $R$ is a *vacuous-orchestration error* if it contains a $(\nu k)$-restricted term with a subterm $R'$ such that

$$R' = \langle k\rangle 1 \mid R'' \quad \text{where } R'' \not\equiv \mathbf{0} \text{ and } R'' \text{ is } a\ k^-\text{-process};$$

v) An *error* is either an orchestrated synchronisation error or a vacuous-orchestration error.

The type system guarantees that a typable process cannot be an error.

**Lemma 4.7.** *Let $\Gamma \Vdash R \triangleright \Delta$. Then $R$ is not an error.*

**Proof.** Immediate by case analysis. □

By means of the Subject Reduction property we show that a typable initial process never reduces to an error.

Concerning the typing of expressions, we assume the standard property that if $\Gamma \vdash e : G$ and $e \downarrow v$ then $\Gamma \vdash v : G$. The following lemma will be used for the Subject Reduction property (Theorem 4.10).

**Lemma 4.8.** *If $\Gamma \vdash e : G$, $\Gamma, x : G \Vdash R \triangleright \Delta$, and $e \downarrow v$ then $\Gamma \Vdash R\{v/x\} \triangleright \Delta$.*

**Proof.** By induction over derivations. □

Before proceeding with the Subject Reduction property, we show that typability is invariant with respect to the structural congruence formalized in Definition 4.1. The presence of orchestrators makes the proof subtler than usual.

**Lemma 4.9.** *If* $\Gamma \Vdash P \rhd \Delta$ *and* $P \equiv Q$ *then* $\Gamma \Vdash Q \rhd \Delta$.

**Proof.** By induction over the definition of $\equiv$. We illustrate the Case 4.1.2, where we have:

$$(\nu k)(\langle k\rangle \mathsf{f} \mid P \mid (\nu k')(\langle k'\rangle \mathsf{g} \mid Q \mid R)) \equiv (\nu k')(\langle k'\rangle \mathsf{g} \mid (\nu k)(\langle k\rangle \mathsf{f} \mid P \mid Q) \mid R)$$

and $k \notin \textsc{fc}(R), k' \notin \textsc{fc}(P)$. The derivation of $\Gamma \Vdash (\nu k)(\langle k\rangle \mathsf{f} \mid P \mid (\nu k')(\langle k'\rangle \mathsf{g} \mid Q \mid R)) \rhd \Delta$ ends by:

$$\dfrac{\dfrac{\Gamma \Vdash P \rhd \Delta_2 \qquad \dfrac{\dfrac{\Gamma \Vdash Q \rhd \Delta_5 \quad \Gamma \Vdash R \rhd \Delta_6}{\Gamma \Vdash Q \mid R \rhd \Delta_4} \qquad \mathsf{g}: S_3 \dashv S_4}{\Gamma \Vdash (\nu k')(\langle k'\rangle \mathsf{g} \mid Q \mid R) \rhd \Delta_3}}{\Gamma \Vdash P \mid (\nu k')(\langle k'\rangle \mathsf{g} \mid Q \mid R) \rhd \Delta_1} \qquad \mathsf{f}: S_1 \dashv S_2}{\Gamma \Vdash (\nu k)(\langle k\rangle \mathsf{f} \mid P \mid (\nu k')(\langle k'\rangle \mathsf{g} \mid Q \mid R)) \rhd \Delta}$$

where $\Delta_1 = \Delta \cdot\ k^- : S_1\ \cdot\ k^+ : S_2$ for some $S_1, S_2$; $\Delta_1 = \Delta_2 \cdot \Delta_3$; $\Delta_4 = \Delta_3 \cdot\ k'^- : S_3\ \cdot\ k'^+ : S_4$ for some $S_3, S_4$; finally $\Delta_4 = \Delta_5 \cdot \Delta_6$. Let's set $\Delta \setminus k = \Delta \setminus \bigcup_{S,S'}\{k^p : S, k^{\overline{p}} : S'\}$. Then such a derivation can be rearranged as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \Vdash P \rhd \Delta_2 \quad \Gamma \Vdash Q \rhd \Delta_5}{\Gamma \Vdash P \mid Q \rhd \Delta_2 \cdot \Delta_5} \qquad \mathsf{f}: S_1 \dashv S_2}{\Gamma \Vdash (\nu k)(\langle k\rangle \mathsf{f} \mid P \mid Q) \rhd (\Delta_2 \cdot \Delta_5) \setminus k} \qquad \Gamma \Vdash R \rhd \Delta_6}{\Gamma \Vdash (\nu k)(\langle k\rangle \mathsf{f} \mid P \mid Q) \mid R \rhd ((\Delta_2 \cdot \Delta_5) \setminus k) \cdot \Delta_6} \qquad \mathsf{g}: S_3 \dashv S_4}{\Gamma \Vdash (\nu k')(\langle k'\rangle \mathsf{g} \mid (\nu k)(\langle k\rangle \mathsf{f} \mid P \mid Q) \mid R) \rhd (((\Delta_2 \cdot \Delta_5) \setminus k) \cdot \Delta_6) \setminus k'}$$

To see that this is a correct derivation, let us assume that $k' \notin dom(\Delta_2)$, because $k' \notin \textsc{fc}(P)$ is the side condition of Case 4.1.2. Now $\Delta_2 \cdot \Delta_5$ is defined since $\Delta_5 \setminus k' = \Delta_3$, and we know that $\Delta_2 \cdot \Delta_3 = \Delta_1$ is defined. On the other hand if $k'' \in dom(\Delta_6)$ then $k'' \notin dom(\Delta_5)$ because $\Delta_5 \cdot \Delta_6 = \Delta_4$ is defined; also if $k'' \neq k$ then $k'' \notin dom(\Delta_2)$ since $dom(\Delta_6) \subseteq dom(\Delta_3) \setminus \{k'\}$ and $\Delta_2$ and $\Delta_3$ are compatible. It follows that $((\Delta_2 \cdot \Delta_5) \setminus k) \cdot \Delta_6$ is defined. Finally, assuming without loss of generality that $k \notin dom(\Delta_6)$ as we know that $k \notin \textsc{fc}(R)$, we conclude:

$$\begin{aligned}
\Delta &= (\Delta_2 \cdot (\Delta_5 \cdot \Delta_6) \setminus k') \setminus k && \text{by the first derivation} \\
&= (\Delta_2 \cdot \Delta_5 \cdot \Delta_6) \setminus k' \setminus k && \text{since } k' \notin dom(\Delta_2) \\
&= ((\Delta_2 \cdot \Delta_5) \setminus k\ \cdot\ \Delta_6 \setminus k) \setminus k' && \\
&= (((\Delta_2 \cdot \Delta_5) \setminus k)\ \cdot\ \Delta_6) \setminus k' && \text{since } k \notin dom(\Delta_6) \qquad \square
\end{aligned}$$

Since no typable processes is an error (Lemma 4.7), the following theorem guarantees (see Corollary 4.11) that no error can appear during the evolution of a typable user-defined process.

**Theorem 4.10** *(Subject reduction). If* $\Gamma \Vdash P \rhd \Delta$ *and* $P \longrightarrow Q$ *then* $\Gamma \Vdash Q \rhd \Delta$.

**Proof.** By induction over the definition of $P \longrightarrow Q$.

In case of rule [Link] we have:

$$\mathtt{request}_S(k)P \mid \mathtt{accept}_{S'}(k)Q \longrightarrow (\nu k)(\langle k\rangle \mathsf{f} \mid P\{k^-/k\} \mid Q\{k^+/k\})$$

for some $\mathsf{f}$ such that $\mathsf{f}: S \dashv S'$. On the other hand, by hypothesis and the shape of the rules, we have the derivation:

$$\dfrac{\dfrac{\Gamma \Vdash P\{k^-/k\} \rhd \Delta \cdot k^- : S}{\Gamma \Vdash \mathtt{request}_S(k)P \rhd \Delta} \qquad \dfrac{\Gamma \Vdash Q\{k^+/k\} \rhd \Delta \cdot k^+ : S'}{\Gamma \Vdash \mathtt{accept}_{S'}(k)Q \rhd \Delta'}}{\Gamma \Vdash \mathtt{request}_S(k)P \mid \mathtt{accept}_{S'}(k)Q \rhd \Delta \cdot \Delta'}$$

where $\Delta \cdot \Delta'$ has to be defined (namely $dom(\Delta) \cap dom(\Delta') = \emptyset$). This implies that:

$$(\Delta \cdot k^- : S) \cdot (\Delta' \cdot k^+ : S') = \Delta \cdot \Delta' \cdot\ k^- : S\ \cdot\ k^+ : S'$$

is defined as well, so that using the fact that $\mathsf{f}: S \dashv S'$, we obtain the following derivation:

$$\dfrac{\dfrac{\Gamma \Vdash P\{k^-/k\} \rhd \Delta \cdot k^- : S \quad \Gamma \Vdash Q\{k^+/k\} \rhd \Delta \cdot k^+ : S'}{\Gamma \Vdash P\{k^-/k\} \mid Q\{k^+/k\} \rhd \Delta \cdot \Delta' \cdot k^- : S \cdot k^+ : S'}}{\Gamma \Vdash (\nu k)(\langle k\rangle \mathsf{f} \mid P\{k^-/k\} \mid Q\{k^+/k\}) \rhd \Delta \cdot \Delta'}$$

In case of rule [OrchComm] we have:

$$(\nu k)(\langle k\rangle \bullet .\mathsf{f} \mid k^p![e].P \mid k^{\overline{p}}?(x).Q) \longrightarrow (\nu k)(\langle k\rangle \mathsf{f} \mid P \mid Q\{v/x\}).$$

Let us suppose without loss of generality that $p = -$ so that $\overline{p} = +$. By hypothesis we have a derivation ending by:

$$\dfrac{\dfrac{\Gamma \Vdash k^-![e].P \rhd \Delta'' \cdot k^- : S \quad \Gamma \Vdash k^+?(x).Q \rhd \Delta''' \cdot k^+ : S'}{\Gamma \Vdash k^-![e].P \mid k^+?(x).Q \rhd \Delta \cdot k^- : S \cdot k^+ : S'}}{\Gamma \Vdash (\nu k)(\langle k\rangle \bullet .\mathsf{f} \mid k^p![e].P \mid k^{\overline{p}}?(x).Q) \rhd \Delta}$$

for some $S, S'$ such that $\bullet.\mathsf{f} : S \dashv S'$, and $\Delta'', \Delta'''$ such that $\Delta = \Delta'' \cdot \Delta'''$. From the derivability of $\Gamma \Vdash k^-![e].P \rhd \Delta'' \cdot k^- : S$ we deduce that $\Gamma \vdash e : G$ for some ground $G$, $\Gamma \Vdash P \rhd \Delta'' \cdot k^- : S''$ and that $S = \mathbf{!}[G].S''$ for some $S''$. Similarly we know that $\Gamma \vdash x : G'$ for some $G'$, $\Gamma, x : G' \Vdash Q \rhd \Delta''' \cdot k^+ : S'''$ and that $S' = \mathbf{?}[G'].S'''$.

Now from this and the fact that $\bullet.\mathsf{f} : \mathbf{!}[G].S'' \dashv \mathbf{?}[G'].S'''$ we infer by Definition 2.3 that $G$ and $G'$ must be the same and that $\mathsf{f} : S'' \dashv S'''$. Hence we have the derivation:

$$\dfrac{\dfrac{\Gamma \Vdash P \rhd \Delta'' \cdot k^- : S'' \quad \Gamma \Vdash Q\{v/x\} \rhd \Delta''' \cdot k^+ : S'''}{\Gamma \Vdash P \mid Q\{v/x\} \rhd \Delta \cdot k^- : S'' \cdot k^+ : S'''}}{\Gamma \Vdash (\nu k)(\langle k\rangle \mathsf{f} \mid P \mid Q\{v/x\}) \rhd \Delta}$$

where $\Gamma \Vdash Q\{v/x\} \rhd \Delta''' \cdot k^+ : S'''$ follows by $\Gamma \vdash e : G$, derivability of $\Gamma, x : G \Vdash Q \rhd \Delta''' \cdot k^+ : S'''$, that $e \downarrow v$ and Lemma 4.8.

In case of [OrchDeleg] we have:

$$(\nu k)(\langle k\rangle \bullet .\mathsf{f} \mid \mathtt{throw}\, k^p[k'^q].P \mid \mathtt{catch}\, k^{\overline{p}}(k').Q) \longrightarrow (\nu k)(\langle k\rangle \mathsf{f} \mid P \mid Q\{k'^q/k'\})$$

Then by hypothesis there exists the derivation:

$$\dfrac{\dfrac{\dfrac{\Gamma \Vdash P \rhd \Delta' \cdot k^p : S_1}{\Gamma \Vdash \mathtt{throw}\, k^p[k'^q].P \rhd \Delta' \cdot k'^q : S' \cdot k^p : \mathbf{!}[S'^q]S_1} \quad \dfrac{\Gamma \Vdash Q\{k'^r/k'\} \rhd \Delta'' \cdot k'^r : S'' \cdot k^{\overline{p}} : S_2}{\Gamma \Vdash \mathtt{catch}\, k^{\overline{p}}(k').Q \rhd \Delta'' \cdot k^{\overline{p}} : \mathbf{?}[S''^r]S_2}}{\Gamma \Vdash \mathtt{throw}\, k^p[k'^q].P \mid \mathtt{catch}\, k^{\overline{p}}(k').Q \rhd \Delta \cdot k'^q : S' \cdot k^p : \mathbf{!}[S'^q]S_1 \cdot k^{\overline{p}} : \mathbf{?}[S''^r]S_2}}{\Gamma \Vdash (\nu k)(\langle k\rangle \bullet .\mathsf{f} \mid \mathtt{throw}\, k^p[k'^q].P \mid \mathtt{catch}\, k^{\overline{p}}(k').Q) \rhd \Delta \cdot k'^q : S'}$$

where $\Delta' \cdot \Delta'' = \Delta$; further, assuming w.l.o.g. that $p = +$ and hence that $\overline{p} = -$, we have $\bullet.\mathsf{f} : \mathbf{!}[S'^q]S_1 \dashv \mathbf{?}[S''^r]S_2$, which implies that $S'' = S'$, $r = q$, and $\mathsf{f} : S_1 \dashv S_2$. Therefore there exists the derivation:

$$\dfrac{\dfrac{\Gamma \Vdash P \rhd \Delta' \cdot k^p : S_1 \quad \Gamma \Vdash Q\{k'^q/k'\} \rhd \Delta'' \cdot k'^q : S' \cdot k^{\overline{p}} : S_2}{\Gamma \Vdash P \mid Q\{k'^q/k'\} \rhd \Delta \cdot k'^q : S' \cdot k^p : S_1 \cdot k^{\overline{p}} : S_2}}{\Gamma \Vdash (\nu k)(\langle k\rangle \mathsf{f} \mid P \mid Q\{k'^q/k'\}) \rhd \Delta \cdot k'^q : S'}$$

where the fact that $\Delta \cdot k'^q : S' \cdot k^p : S_1 \cdot k^{\overline{p}} : S_2$ is well defined follows by the fact that $\Delta \cdot k'^q : S' \cdot k^p : \mathbf{!}[S'^q]S_1 \cdot k^{\overline{p}} : \mathbf{?}[S''^r]S_2$ is such.

Let us consider the case [OrchSSel]:

$$(\nu k)(\langle k\rangle \oplus_{h\in H} l_h.\mathsf{f}_h \mid k^p \lhd [l_j : P_j]_{j\in J} \mid k^{\overline{p}} \rhd \{l_i : Q_i\}_{i\in I}) \longrightarrow (\nu k)(\langle k\rangle \mathsf{f}_c \mid P_c \mid Q_c)$$

where $c \in H \cup I \cup J$. By hypothesis we have a derivation ending by:

$$\dfrac{\Gamma \Vdash k^p \lhd [l_j : P_j]_{j\in J} \mid k^{\overline{p}} \rhd \{l_i : Q_i\}_{i\in I} \rhd \Delta \cdot k^p : \boxplus\{l_j : S_j\}_{j\in J} \cdot k^{\overline{p}} : \&\{l_i : S'_i\}_{i\in I}}{\Gamma \Vdash (\nu k)(\langle k\rangle \oplus_{h\in H} l_h.\mathsf{f}_h \mid k^p \lhd [l_j : P_j]_{j\in J} \mid k^{\overline{p}} \rhd \{l_i : Q_i\}_{i\in I}) \rhd \Delta}$$

with the side condition $\oplus_{h\in H} l_h.\mathsf{f}_h : \boxplus\{l_j : S_j\}_{j\in J} \dashv \&\{l_i : S'_i\}_{i\in I}$. By Definition 2.3 this implies that $H \subseteq J \cap I$ is a non empty set such that for all $h \in H$ it holds $\mathsf{f}_h : S_h \dashv S'_h$. The premise of the last inference in the above derivation must be derived by rule [Conc-T] from

$$\dfrac{\forall j \in J \quad \Gamma \Vdash P_j \rhd \Delta' \cdot k^p : S_j}{\Gamma \Vdash k^p \lhd [l_j : P_j]_{j\in J} \rhd \Delta' \cdot k^p : \boxplus\{l_j : S_j\}_{j\in J}} \quad \text{and} \quad \dfrac{\forall i \in I \quad \Gamma \Vdash Q_i \rhd \Delta'' \cdot k^{\overline{p}} : S'_i}{\Gamma \Vdash k^{\overline{p}} \rhd \{l_i : Q_i\}_{i\in I} \rhd \Delta'' \cdot k^{\overline{p}} : \&\{l_i : S'_i\}_{i\in I}}$$

where $\Delta = \Delta' \cdot \Delta''$ is defined. Now since $c \in H \subseteq J \cap I$, we know that $\mathsf{f}_c : S_c \dashv S'_c$ so that from the above we obtain the derivation:

$$\dfrac{\dfrac{\Gamma \Vdash P_c \rhd \Delta' \cdot k^p : S_c \quad \Gamma \Vdash Q_c \rhd \Delta'' \cdot k^{\overline{p}} : S'_c}{\Gamma \Vdash P_c \mid Q_c \rhd \Delta \cdot k^p : S_c \cdot k^{\overline{p}} : S'_c}}{\Gamma \Vdash (\nu k)(\langle k \rangle \mathsf{f}_c \mid P_c \mid Q_c) \rhd \Delta}$$

The case of rule [OrchSel] is similar and simpler. Finally rules [Par], [Scop] and [Str] follow by induction and Lemma 4.9. □

**Corollary 4.11** *(Type safety). If $P$ is a user-defined process such that $\Gamma \Vdash P \rhd \Delta$ for some $\Gamma$ and $\Delta$, and $R$ is a run-time process such that $P \xrightarrow{*} R$, then $R$ is not an error.*

Notice that, whereas our type system prevents deadlocks like the last one in Example 4.5, it cannot prevent deadlocks due to the presence of subterms like $(\nu k)(k^+![e].R \mid R' \mid \langle k \rangle \mathtt{1})$, where $R'$ is not a $k^-$- process (see Example 5.1 and Definition 5.2 below). This sort of deadlocks is intrinsically due to the asymmetric nature of our compliance relation. They can be avoided either by using a less general compliance relation (namely a symmetric restriction of the present one), or by extending the operational semantics, as we shall do in the next section.

## 5. Compliance-dependent deadlocks and clean-up reductions

In the present setting compatibility is asymmetric, since the client is allowed to leave a session as soon as it reaches a successful state. This implies that even well typed, and hence well behaved pairs of processes may end up in a state where actions on the server side remain unmatched. In Example 4.4, if the client connects to *ProviderII* in order to rent a movie and terminates after the reception of a no message, the output action of *ProviderII* justifying the refusal of the rental remains pending. In some cases this sort of behaviour could cause some problems, as shown in the following example.

**Example 5.1.** Let us consider the following processes:

$$P = \mathtt{request}_{\mathsf{S}_a}(k)\,\mathtt{request}_{\mathsf{S}_b}(k')\; k![4].\,k'![\mathrm{True}]$$
$$Q = \mathtt{accept}_{\mathsf{S}_c}(k)\,\mathtt{accept}_{\mathsf{S}_d}(k')\; k?(x).\,k![x].\,k'?(y)$$

where $\mathsf{S}_a = \mathbf{!}[\mathtt{Nat}]$, $\mathsf{S}_b = \mathbf{!}[\mathtt{Bool}]$, $\mathsf{S}_c = \mathbf{?}[\mathtt{Nat}].\mathbf{!}[\mathtt{Nat}]$, $\mathsf{S}_d = \mathbf{?}[\mathtt{Bool}]$. By the typing rules in Fig. 2 we derive:

$$\Vdash k^-![4].\,k'^-![\mathrm{True}] \rhd k^- : \mathsf{S}_a, k'^- : \mathsf{S}_b$$
$$\Vdash k^+?(x).\,k^+![x].\,k'^+?(y) \rhd k^+ : \mathsf{S}_c, k'^+ : \mathsf{S}_d$$

and therefore both $\Vdash P \rhd \emptyset$ and $\Vdash Q \rhd \emptyset$ by rules Req-T and Acc-T respectively, which implies $\Vdash P \mid Q \rhd \emptyset$ by Conc-T. By definition of the orchestrated compliance relation in Fig. 1, we have that both $\bullet.\mathtt{1} : \mathsf{S}_a \dashv \mathsf{S}_c$ and $\bullet.\mathtt{1} : \mathsf{S}_b \dashv \mathsf{S}_d$. It follows that

$$\begin{array}{ll} & P \mid Q \\ \xrightarrow{*} & (\nu k')(\,\langle k' \rangle \bullet.\mathtt{1} \mid (\nu k)(\,\langle k \rangle \bullet.\mathtt{1} \mid k^-![4].\,k'^-![\mathrm{True}] \mid k^+?(x).\,k^+![x].\,k'^+?(y))) \\ \longrightarrow & (\nu k')(\,\langle k' \rangle \bullet.\mathtt{1} \mid (\nu k)(\,\langle k \rangle \mathtt{1} \mid k'^-![\mathrm{True}] \mid k^+![4].\,k'^+?(y))) \\ \equiv & (\nu k')(\,\langle k' \rangle \bullet.\mathtt{1} \mid (\nu k)(\,\langle k \rangle \mathtt{1} \mid k^+![4].\,k'^+?(y) \mid k'^-![\mathrm{True}])) \\ \not\longrightarrow & \end{array}$$

This is not a counterxample to Corollary 4.11 since $k^+![4].k'^+?(y)$ and $k'^-![\mathrm{True}]$ are a $k$-process and a $k'$-process respectively, but for *distinct* $k$ and $k'$, hence not an error according to Definition 4.6.

The situation illustrated in the example above resembles a deadlock, e.g.:

$$(\nu k')(\,\langle k' \rangle \bullet.\mathtt{1} \mid (\nu k)(\,\langle k \rangle \bullet.\mathtt{1} \mid k'^-?(x).\,k^-![4] \mid k^+?(y).\,k'^+![\mathrm{True}])) \quad (1)$$

Again this is not an error, but unfortunately it typechecks in our system, much as it is the case for standard session type systems, as e.g. [6]. To ensure deadlock freeness or the weaker property of progress of typable terms, systems are equipped with sophisticated annotations to check against circularities in channel dependancies (see e.g. [18], §7.5 for references to the literature), as it happens with channels $k$ and $k'$ in (1). However the problem illustrated in Example 5.1 is not caused

[OrchClnUp1]
$(\nu k)(\langle k\rangle \mathtt{1} \mid \pi.R' \mid R) \longrightarrow (\nu k)(\langle k\rangle \mathtt{1} \mid R' \mid R)$
where $\pi \in \{k^+![e], k^+?(x), \mathtt{throw}\, k^+[k'^q], \mathtt{catch}\, k^+(k'), k^+ \triangleleft l\}$

[OrchClnUp2]
$(\nu k)(\langle k\rangle \mathtt{1} \mid k^+ \triangleright \{l_i{:}R'_i\}_{i\in I} \mid R) \longrightarrow (\nu k)(\langle k\rangle \mathtt{1} \mid R'_c \mid R)$ if $c \in I$

[OrchClnUp3]
$(\nu k)(\langle k\rangle \mathtt{1} \mid k^+ \triangleleft [l_i{:}R'_i]_{i\in I} \mid R) \longrightarrow (\nu k)(\langle k\rangle \mathtt{1} \mid R'_c \mid R)$ if $c \in I$

**Fig. 5.** Clean-up reductions.

by any kind of circularity, and would be not fixed by moving to such complex extensions of session types; rather it clearly depends on the choice of using compliance in place of duality when matching a client with a server, since after the client has abandoned a session, the code of session continuation remains on the server side. This does not happen with ordinary session types where processes like that in Example 5.1 do not typecheck. We call *compliance-dependent deadlock* such a stuck state.

**Definition 5.2.** A process $P$ is a *compliance-dependent deadlock* if $P \not\longrightarrow$ and it contains a subterm $R$ of the form

$$R = (\nu k)(\langle k\rangle \mathtt{1} \mid R' \mid R'') \quad \text{where } R' \not\equiv \mathbf{0} \text{ is a } k^+\text{-process.}$$

Existence of such a kind of deadlock hinders the application of the concept of compliance from contract theory, since a client should be allowed to abandon a session even before the process on the server side has completed, without compromising further interactions within different sessions.

As it is clear from the definition, in a compliance-dependent deadlock the block is caused by the unmatched communications on the server side using the channel $k$ that is marked as closed by the named orchestrator $\langle k\rangle \mathtt{1}$, i.e. the orchestrator that successfully completed all the actions requested by the client on channel $k$. To face the problem, instead of changing the type system we consider an extension to the operational semantics as in Fig. 5, formalizing a form of abort on the server side by means of reduction rules that we call **clean-up reductions**.

**Remark 5.3.** When adding recursion to the calculus with clean-up reductions (for instance by the means of a term-variable binder $\mu$) the equi-recursive approach usually considered with session type systems (since [5]) has to be abandoned in favour of explicit fold/unfold term constructors. Otherwise say $(\nu k)(\langle k\rangle \mathtt{1} \mid \mu X.k?(x).X \mid \mathbf{0})$ reduces to itself by one step of clean-up reduction since then $\mu X.k?(x).X \equiv k?(x).(\mu X.k?(x).X)$.

By adding clean-up reductions to the operational semantics the subject reduction property keeps on holding without any change to the type system.

**Theorem 5.4** *(Subject reduction with clean-up).*
*Let $\longrightarrow$ be the reduction relation defined by the rules of Figs. 3 and 5.*
*If $\Gamma \Vdash P \triangleright \Delta$ and $P \longrightarrow Q$ then $\Gamma \Vdash Q \triangleright \Delta$.*

**Proof.** The proof can be obtained by extending the proof of Theorem 4.10 with the cases corresponding to the rules of Fig. 5.

In case of rule [OrchClnUp1], with $\pi = k^+![e]$, we have

$$(\nu k)(\langle k\rangle \mathtt{1} \mid k^+![e].R' \mid R) \longrightarrow (\nu k)(\langle k\rangle \mathtt{1} \mid R' \mid R)$$

and hence the derivation has the following form

$$\dfrac{\dfrac{\Gamma \Vdash k^+![e].R' \triangleright \Delta_1 \cdot k^+ : ![G].S' \qquad \Gamma \Vdash R \triangleright \Delta_2 \cdot k^- : S}{\Gamma \Vdash k^+![e].R' \mid R \triangleright \Delta_1 \cdot \Delta_2 \cdot k^- : S \cdot k^+ : ![G].S'} \qquad \mathtt{1} : S \dashv ![G].S'}{\Gamma \Vdash (\nu k)(\langle k\rangle \mathtt{1} \mid k^+![e].R' \mid R) \triangleright \Delta_1 \cdot \Delta_2 = \Delta}$$

By definition of compliance, it immediately follows that $S = \mathsf{end}$. Now, since from the above we can also get that $\Gamma \Vdash R' \triangleright \Delta_1 \cdot k^+ : S'$, it is possible to build the following derivation:

$$\dfrac{\dfrac{\Gamma \Vdash R' \triangleright \Delta_1 \cdot k^+ : S' \qquad \Gamma \Vdash R \triangleright \Delta_2 \cdot k^- : \mathsf{end}}{\Gamma \Vdash R' \mid R \triangleright \Delta_1 \cdot \Delta_2 \cdot k^- : \mathsf{end} \cdot k^+ : S'} \qquad \mathtt{1} : \mathsf{end} \dashv S'}{\Gamma \Vdash (\nu k)(\langle k\rangle \mathtt{1} \mid R' \mid R) \triangleright \Delta_1 \cdot \Delta_2 = \Delta}$$

All the other cases for $\pi$, as well as the cases for the other two clean-up rules, can be treated in a similar way. □

**Corollary 5.5** *(Type safety w.r.t. clean-up). Let $\longrightarrow$ be the reduction relation defined by the rules in Fig. 3 and 5.*

*If $P$ is a user-defined process such that $\Gamma \Vdash P \triangleright \Delta$ for some $\Gamma$ and $\Delta$, and $R$ is a run-time process such that $P \xrightarrow{*} R$ then $R$ is neither an error nor a compliance-dependent deadlock.*

## 6. Subtyping and orchestrator functors

Subtyping polymorphism is a common technique to make type systems more expressive and flexible. In this section we study subtyping in our system, where it is orthogonal with respect to retractable compliance and orchestrators. Indeed, much as with type systems for object-oriented languages, subtyping deals with specialization of software modules, while keeping a more general interface. Suppose that a service is delivered in sessions of type $S$, but its actual implementation, that might well change in time, has type $S'$, where $S'$ is a supertype of $S$. In usual systems this suffices to conclude that any client complying with a service of type $S$ will do the same with its implementation of type $S'$. But since we have relaxed the concept of compliance by means of orchestrators, it is questionable whether the same orchestrator will do the job with actual implementations of services, and we readily realize that this is not the case. A similar issue arises on the client side, as some new version of a client might improve on an older one, yielding a process typable by some supertype of the previous version. The new version is supposed nonetheless to behave properly w.r.t. those services with which the older one was compliant.

### *6.1. Introductory discussion and the need of orchestrator functors*

According to [17] the idea of subtyping for mobile processes coming from [19] can be adapted to the case of session types as follows:

> if $S_1$ is a subtype of $S_2$ then a session channel of type $S_1$ can safely be used anywhere that a session channel of type $S_2$ is expected.

In [17] the authors were using polarised channel names just to pair the two end-points of the very same channel, without assigning any other meaning to polarities. Instead, in the present setting the negative polarity in $k^-$ refers to the client's end-point, while the positive polarity in $k^+$ refers to the server's end-point of the channel $k$. Since we are replacing type duality, a symmetric relation, by the non-symmetric relation of orchestrated-compliance, two notions of "subtyping" have to be introduced.

Following ideas in [7], extensively investigated e.g. in [12] and [4], we get two subtyping relations that are tentatively defined by

$$\begin{array}{lcl} S \preccurlyeq_+ S' & \triangleq & \forall S''.(\exists \mathsf{f}.\, \mathsf{f} : S'' \dashv S \implies \exists \mathsf{f}'.\, \mathsf{f}' : S'' \dashv S') \\ S \preccurlyeq_- S' & \triangleq & \forall S''.(\exists \mathsf{f}.\, \mathsf{f} : S \dashv S'' \implies \exists \mathsf{f}'.\, \mathsf{f}' : S' \dashv S'') \end{array} \qquad (2)$$

Spelling this in words, if we identify clients and servers with the types labelling the user-defined processes $\mathtt{request}_S(k)P$ and $\mathtt{accept}_S(k)P$ respectively, then $\mathsf{Client}(S) = \{S'' \mid \exists \mathsf{f} : S'' \dashv S\}$ is the set of all "clients" compliant with the "server" $S$, and similarly $\mathsf{Server}(S) = \{S'' \mid \exists \mathsf{f} : S \dashv S''\}$ is the set of all "servers" compliant with the "client" $S$. Then $S \preccurlyeq_+ S'$ holds if $\mathsf{Client}(S) \subseteq \mathsf{Client}(S')$, and $S \preccurlyeq_- S'$ if $\mathsf{Server}(S) \subseteq \mathsf{Server}(S')$.

These subtyping relations could hence be used in our formalism to obtain a more flexible type system, for example by extending rules [Acc-T] and [Req-T] as follows.

$$[\text{Acc-T-}\preccurlyeq]\ \frac{\Gamma \Vdash P\{k^+/k\} \triangleright \Delta \cdot k^+ : S' \qquad S \preccurlyeq_+ S'}{\Gamma \Vdash \mathtt{accept}_S(k)P \triangleright \Delta}$$

$$[\text{Req-T-}\preccurlyeq]\ \frac{\Gamma \Vdash P\{k^-/k\} \triangleright \Delta \cdot k^- : S' \qquad S \preccurlyeq_- S'}{\Gamma \Vdash \mathtt{request}_S(k)P \triangleright \Delta}$$

By the latter rule, process $P\{k^-/k\}$ is able to interact (as a client) according to the protocol described by $S'$. Since, by $S \preccurlyeq_- S'$, any server for a client behaving as $S$ is also a server for a client behaving as $S'$, it is safe to declare process $\mathtt{request}_S(k)P$ to be a client willing to interact on a session channel according to type $S$. A dual argument applies for the former rule.

However, in the typing rules above, even if they look quite natural, there is an aspect which has been overlooked: the $\mathsf{f}$ and $\mathsf{f}'$ in (2) above can be different, and in general are so, as it is shown by the following example.

**Example 6.1.** Let us define

$$\begin{aligned} \mathsf{A} &= \mathbf{!}[\texttt{Nat}].\&\{l_1{:}\mathbf{!}[\texttt{Bool}], l_2{:}\mathbf{?}[\texttt{Bool}], l_3 : \mathsf{end}, l_4 :\mathbf{?}[\texttt{Nat}]\} \\ \mathsf{B} &= \mathbf{?}[\texttt{Nat}].\boxplus\{m{:}\mathbf{?}[\texttt{Nat}], l_2{:}\mathbf{!}[\texttt{Bool}].\mathbf{!}[\texttt{Nat}], l_3 :\mathbf{?}[\texttt{Nat}]\} \\ \mathsf{B}' &= \mathbf{?}[\texttt{Nat}].\bigoplus\{l_1{:}\mathbf{!}[\texttt{Bool}], l_2{:}\mathbf{!}[\texttt{Bool}], l_3 :\mathbf{?}[\texttt{Nat}]\} \end{aligned}$$

It is not difficult to check that the process

$$Q = \texttt{accept}_{\mathsf{B}'}(k)k?(y).k \triangleleft [m{:}k?(x), l_2{:}k![\mathsf{tt}].k![y], l_3 : k?(z)]$$

is typable, since $k?(y).k \triangleleft [m{:}k?(x), l_2{:}k![\mathsf{tt}].k![y], l_3 : k?(z)]$ is typable using the typing $k^+ : \mathsf{B}$, and rule [Acc-T] applies because of $\mathsf{B}' \preccurlyeq_+ \mathsf{B}$.

On the other hand it is possible to check that the process

$$P = \texttt{request}_{\mathsf{A}}(k)k![5].k \triangleright \{l_1{:}k![3], l_2{:}k?(x), l_3 : \mathbf{0}, l_4 : k?(z)\}$$

is typable by means of rule [Req-T] since, trivially, $\mathsf{A} \preccurlyeq_- \mathsf{A}$. Then the system $P \mid Q$ is typeable, so that a session can be established between $P$ and $Q$, and indeed $\mathsf{A} \dashv \mathsf{B}'$.

The evolution of $P \mid Q$, however, gets stuck after a few reduction steps, as shown below.

$$\begin{aligned} & P \mid Q \\ & \longrightarrow \\ & (\nu k)(\langle k\rangle \bullet .(l_1.\bullet + l_2.\bullet + l_3.\bullet) \mid k![5].k \triangleright \{l_1{:}k![\mathsf{tt}].k![3], l_2{:}k?(x), l_3{:}\mathsf{end}, l_4{:}k?(z)\} \mid \mathsf{Q}') \\ & \text{where } \mathsf{Q}' = k?(y).k \triangleleft [m{:}k?(x), l_2{:}k![\mathsf{tt}].k![y], l_3{:}k?(z)] \\ & \longrightarrow \\ & (\nu k)(\langle k\rangle(l_1.\bullet + l_2.\bullet + l_3.\bullet) \mid k \triangleright \{l_1{:}k![\mathsf{tt}].k![3], l_2{:}k?(x), l_3 : \mathsf{end}, l_4{:}k?(z)\} \mid \mathsf{Q}'') \\ & \text{where } \mathsf{Q}'' = k \triangleleft [m{:}k?(x), l_2{:}k![\mathsf{tt}].k![y], l_3{:}k?(z)] \\ & \not\longrightarrow \end{aligned}$$

Here the system gets stuck since the speculative choice requires an orchestrator of the form $\oplus l_i.f_i$, whereas we have here one of the form $\sum l_i.f_i$.

This problem arises since, by means of the subtyping relation, we have labelled the `request` operator with $\mathsf{B}'$ instead of $\mathsf{B}$. Hence, the orchestrator introduced in the system by the first reduction is $\bullet.(l_1.\bullet + l_2.\bullet + l_3.\bullet)$ (which is also the only possible one for an interaction between $\mathsf{A}$ and $\mathsf{B}'$). For a correct interaction, instead, the orchestrator $\bullet.(l_1.\bullet \oplus l_2.\bullet)$ (or $l_1.\bullet$, or $l_2.\bullet$) would have been necessary.

A way to recover from the above problem lays on the idea underlying the calculi with *explicit-subtyping*, where subtyping judgements come equipped with explicit coercion functions. Explicit subtyping has been widely investigated in several settings, as object-oriented languages and type theories (see e.g. [20] and [21]).

In particular, we equip a subtyping judgement like $\mathsf{B}' \preccurlyeq_+ \mathsf{B}$ with an *orchestrator-functor* $F$ such that, for any $\mathsf{C}$ and $\mathsf{f}$, if $\mathsf{f} : \mathsf{C} \dashv \mathsf{B}'$ then $F(\mathsf{f}) : \mathsf{C} \dashv \mathsf{B}$; whereas a subtyping judgement $\mathsf{D}' \preccurlyeq_- \mathsf{D}$ is equipped with an orchestrator-functor $F$ such that, for any $\mathsf{C}$ and $\mathsf{f}$, if $\mathsf{f} : \mathsf{D}' \dashv \mathsf{C}$ then $F(\mathsf{f}) : \mathsf{D} \dashv \mathsf{C}$. (A similar idea has been carried out in [22] in the context of retractable session contracts, where subcontract derivations are interpreted as orchestrator functors).

The problem of the Example 6.1 above would hence be solved by means of a functor $F$ such that $F(\bullet.(l_1.\bullet + l_2.\bullet + l_3.\bullet)) = \bullet.(l_1.\bullet \oplus l_2.\bullet)$. Such a functor should be obtainable out of the functors $F_1$ and $F_2$ which equip, respectively, the subtyping judgements $\mathsf{B}' \preccurlyeq_+ \mathsf{B}$ and $\mathsf{A} \preccurlyeq_- \mathsf{A}$. This implies that these two functors should be part, respectively, of the `request` and `accept` operators (see Definitions 6.15 and 6.18).

As we shall make clearer below, functors are not just useful when a session is established, as they are also needed to correctly deal with the catch/throw mechanism. The resulting system is technically complex, but still convenient from the programmer's point of view. Indeed orchestrator functors don't need to be specified by hand; they can be automatically computed, by implementing the syntax-directed system in Fig. 6. This happens simultaneously with the check of the subtyping relation, without computational overhead. As such they are transparent to programmers, who are given the double advantage of avoiding tedious coding of exception handlers on the one hand, and of a handy programming construct for coding choice preferences on the other.

*6.1.1. Orchestrator functors and their operational semantics*

First we extend the syntax of orchestrators.

**Definition 6.2** *(Orchestrators with subtyping).* We define the set $\mathsf{Orch}_{\preccurlyeq}$ of *orchestrators with subtyping* by adding to Definition 2.2 the following production

$$\mathsf{f}, \mathsf{g} ::= \cdots \mid \bullet_F.\mathsf{f} \qquad \text{higher-order I/O prefix}$$

where $F \in \mathsf{OrchF}$ (see Definition 6.3 below).

The orchestrator action $\bullet_F$ is used to mediate catch/throw synchronizations. The functor $F$ has the aim of properly modifying the orchestrator of the delegated channel-end.

**Definition 6.3** *(Orchestrator functors).* Let $\varphi$ range over a set of orchestrator variables. We define $\mathsf{OrchF}$, the set of orchestrator functors $F$, by the following grammar.

$$\begin{aligned} F \;&:= \mid \lambda\varphi.\mathbf{1} \\ &\quad \mid \lambda\langle\bullet\rangle.F \mid \lambda\langle\bullet^{\mathrm{L}}_F\rangle.F \mid \lambda\langle\bullet^{\mathrm{R}}_F\rangle.F \\ &\quad \mid \lambda\langle\odot,\odot'\rangle.\{F_i\}_{i\in I} \\ \odot \;&:= \oplus \mid + \end{aligned}$$

We call *functor application* an element of $\mathsf{OrchF} \times \mathsf{Orch}$. We shall denote functor applications by expressions of the form $F(\mathsf{f})$, where $F \in \mathsf{OrchF}$ and $\mathsf{f} \in \mathsf{Orch}$.

Before providing an operational semantics for functors, we define a composition operator.

As it will be apparent later on, we need composition of functors in order to deal with session-opening synchronizations (see Remark 6.17) and delegation (see Example 6.19).

Notice that (as it will be clarified in Remark 6.7) we define the composition operator in an explicit way, without using the notion of functor application.

**Definition 6.4** *(Functor sequential composition).*

The sequential-composition operator on functors

$$\mathbf{;} : \mathsf{OrchF} \times \mathsf{OrchF} \longrightarrow \mathsf{OrchF} \cup \{\text{✗}\}$$

is defined by induction on the structure of its arguments, as follows.

$$\begin{aligned}
\lambda\varphi.\mathbf{1}\,;G &\triangleq \lambda\varphi.\mathbf{1} \\
(\lambda\langle\bullet\rangle.F)\,;(\lambda\langle\bullet\rangle.G) &\triangleq \lambda\langle\bullet\rangle.(F\,;G) \\
(\lambda\langle\bullet^{\mathrm{L}}_{F_1}\rangle.F_2)\,;(\lambda\langle\bullet^{\mathrm{L}}_{G_1}\rangle.G_2) &\triangleq \lambda\langle\bullet^{\mathrm{L}}_{F_1\,;\,G_1}\rangle.(F_2\,;G_2) \\
(\lambda\langle\bullet^{\mathrm{R}}_{F_1}\rangle.F_2)\,;(\lambda\langle\bullet^{\mathrm{R}}_{G_1}\rangle.G_2) &\triangleq \lambda\langle\bullet^{\mathrm{R}}_{G_1\,;\,F_1}\rangle.(F_2\,;G_2) \\
(\lambda\langle+,+,X\rangle.\{F_x\}_{x\in X})\,;(\lambda\langle+,+,X\rangle.\{F_x\}_{x\in X}) &\triangleq \lambda\langle+,+,X\rangle.\{F_x\}_{x\in X} \\
(\lambda\langle\oplus,\oplus,X\rangle.\{F_x\}_{x\in X})\,;(\lambda\langle\oplus,\oplus,X\rangle.\{F_x\}_{x\in X}) &\triangleq \lambda\langle\oplus,\oplus,X\rangle.\{F_x\}_{x\in X} \\
(\lambda\langle+,+,X\rangle.\{F_j\}_{j\in J})\,;(\lambda\langle+,\oplus,X\rangle.\{F_k\}_{k\in K}) &\triangleq \lambda\langle+,\oplus,X\rangle.\{F_k\}_{k\in K} \\
(\lambda\langle+,+,X\rangle.\{F_j\}_{j\in J})\,;(\lambda\langle+,+,X\rangle.\{F_k\}_{k\in K}) &\triangleq \lambda\langle+,+,X\rangle.\{F_k\,;G_k\}_{k\in K} \\
(\lambda\langle+,\oplus,X\rangle.\{F_k\}_{k\in K})\,;(\lambda\langle\oplus,\oplus,X\rangle.\{F_x\}_{x\in X}) &\triangleq \lambda\langle+,\oplus,X\rangle.\{F_k\}_{k\in K} \\
F\,;G &\triangleq \text{✗} \qquad \text{in all other cases}
\end{aligned}$$

We say that $F\,;G$ is *defined* whenever $F\,;G \neq$ ✗, that is $F\,;G \in \mathsf{OrchF}$.

At a glance, the expression $\lambda\varphi.\mathbf{1}\,;G$ in the first item of the above definition looks like a typo: one would intuitively expect it to be $G\,;\lambda\varphi.\mathbf{1}$. The first item has that form since, according to the operational semantics below, the application of any functors to the orchestrator $\mathbf{1}$ yields $\mathbf{1}$.

We define now the operational semantics of orchestrator functors by means of the following reduction relation between functor applications and orchestrators.

**Definition 6.5** *(Functor-application reduction).*

The computational behaviour of functors is defined by the reduction relation $\longrightarrow$, where

$$\longrightarrow \subseteq (\mathsf{OrchF} \times \mathsf{Orch}) \times \mathsf{Orch}$$

which is defined by the compatible closure of the following notions of reductions:

- $F(\mathtt{1}) \longrightarrow \mathtt{1}$
- $(\lambda\varphi.\mathtt{1})(\mathsf{f}) \longrightarrow \mathtt{1}$
- $(\lambda\langle\bullet\rangle.F)(\bullet.\mathsf{f}) \longrightarrow \bullet.(F(\mathsf{f}))$
- $(\lambda\langle\bullet^{\mathrm{L}}_{G}\rangle.F)(\bullet_{F'}.\mathsf{f}) \longrightarrow \bullet_{F'\,;\,G}.(F(\mathsf{f}))$ if $F'\,;\,G$ is defined.
- $(\lambda\langle\bullet^{\mathrm{R}}_{G}\rangle.F)(\bullet_{F'}.\mathsf{f}) \longrightarrow \bullet_{G\,;\,F'}.(F(\mathsf{f}))$ if $G\,;\,F'$ is defined.
- $(\lambda\langle\odot,\odot',X\rangle.\{F_i\}_{i\in S(X)})(\odot_{k\in K} l_k.\mathsf{f}_k) \longrightarrow \odot'_{h\in\{K/X\}S(X)} l_h.(F_h(\mathsf{f}_h))$ *if* $\{K/X\}S(X) \subseteq K$

where

- the variable $X$ ranges over sets of indexes and it is bound in $\lambda\langle\odot,\odot',X\rangle.\{F_i\}_{i\in S(X)}$;
- $S(X)$ is an expression formed using the set operators on set of indexes and, possibly, the variable $X$;
- $\{K/X\}S(X)$ denotes the set corresponding to the expression $S(X)$ where $X$ is replaced by the set $K$.

We write $\textsc{nf}(F(\mathsf{f}))$ to denote the normal form of $F(\mathsf{f})$. When clear from the context, we shall simply write $F(\mathsf{f})$ for $\textsc{nf}(F(\mathsf{f}))$.

We can now check that $F\,;\,G$ does behave as a sequential composition.

**Lemma 6.6.** *Let $F, G \in \mathsf{OrchF}$ and $\mathsf{f} \in \mathsf{Orch}$. If $F\,;\,G$ is defined, and $F(\mathsf{f}) \in \mathsf{Orch}$ then*

$$(F\,;\,G)(\mathsf{f}) = G(F(\mathsf{f}))$$

**Proof.** By induction on the structure of $F$ and $G$, using Definition 6.5. □

**Remark 6.7.** A more standard definition of $F\,;\,G$ would be of course $\lambda\varphi.G(F(\varphi))$. This has however the technical disadvantage of introducing a full λ-calculus of orchestrator functors whose application and properties need further investigation. We prefer instead the explicit Definition 6.4 that suffices for our purposes.

A functor is used to *reconstruct* the orchestrator given as input, modifying it during the reconstruction. In particular, some choices are possibly restricted, and some $+$ and $\oplus$ operators are possibly replaced by with other ones. In fact, it is easy to check that:

$$\begin{array}{ll} & F(\bullet.\sum_{i\in\{1,2,3\}} l_i.\bullet) \\ \longrightarrow & \bullet.(\lambda\langle +,\oplus,X\rangle.\{F_i\}_{i\in\{1,2,3\}})(\sum_{i\in\{1,2\}} l_i.\bullet) \\ \longrightarrow & \bullet.\bigoplus_{i\in\{1,2\}} F_i(\bullet) \\ \longrightarrow^2 & \bullet.\bigoplus_{i\in\{1,2\}} \bullet.(\lambda\varphi.\mathtt{1})(\mathtt{1}) \\ \longrightarrow^2 & \bullet.\bigoplus_{i\in\{1,2\}} \bullet.\mathtt{1} \end{array}$$

**Example 6.8.** A functor for Example 6.1 is described by the expression

$$F = \lambda\langle\bullet\rangle.(\lambda\langle +,\oplus,X\rangle.\{F_i\}_{i\in\{1,2\}})$$

where $F_1 = F_2 = (\lambda\langle\bullet\rangle.\lambda\varphi.\mathtt{1})$, and it is such that:

$$F(\bullet.(l_1.\bullet + l_2.\bullet + l_3.\bullet)) \longrightarrow^* \bullet.(l_1.\bullet \oplus l_2.\bullet) \qquad (3)$$

By Definitions 6.3 and 6.5, in case of a reconstruction of an orchestrator containing a higher-order action labelled by $F'$, this is replaced by the left (L) or right (R) composition of $F'$ with the functor $G$ present as label in the functor abstraction.

*6.1.2. Witnessed subtyping*

The next step is a formal definition of the statement that $S \preccurlyeq_p S'$, where $p$ is a polarity, up to an orchestrator functor $F$ witnessing the subtyping relation. Following [12] and [13], we introduce also a third subtyping relation, the *peer*-subtyping $\preccurlyeq_{\pm}$, meaning that there is no bias towards the client.

$$[\text{Ax - }\preccurlyeq_+]\ \lambda\varphi.\mathtt{1} : \mathsf{end} \preccurlyeq_+ S \qquad [\text{Ax - }\preccurlyeq_\pm]\ \lambda\varphi.\mathtt{1} : \mathsf{end} \preccurlyeq_\pm \mathsf{end} \qquad [\text{Ax - }\preccurlyeq_-]\ \lambda\varphi.\mathtt{1} : S \preccurlyeq_- \mathsf{end}$$

$$[\text{Val-Input - }\preccurlyeq_p]\ \frac{F : S \preccurlyeq_p S'}{\lambda\langle\bullet.\varphi\rangle.\bullet.F : \mathbf{?}[G].S \preccurlyeq_p \mathbf{?}[G].S'} \qquad [\text{Val-Output - }\preccurlyeq_p]\ \frac{F : S \preccurlyeq_p S'}{\lambda\langle\bullet.\varphi\rangle.\bullet.F : \mathbf{!}[G].S \preccurlyeq_p \mathbf{!}[G].S'}$$

$$[\text{S-Input - }\preccurlyeq_p]\ \frac{F_1 : S_1 \preccurlyeq_q S'_1 \quad F_2 : S_2 \preccurlyeq_p S'_2}{\lambda\langle\bullet^{\mathsf{L}}_{F_1}\rangle.F_2 : \mathbf{?}[S^q_1].S_2 \preccurlyeq_p \mathbf{?}[S'^q_1].S'_2} \qquad [\text{S-Output - }\preccurlyeq_p]\ \frac{F_1 : S'_1 \preccurlyeq_q S_1 \quad F_2 : S_2 \preccurlyeq_p S'_2}{\lambda\langle\bullet^{\mathsf{R}}_{F_1}\rangle.F_2 : \mathbf{!}[S^q_1].S_2 \preccurlyeq_p \mathbf{!}[S'^q_1].S'_2}$$

$$[\&\cdot\&\text{ - }\preccurlyeq_p]\ \frac{\forall i \in I \subseteq J \quad F_i : S_i \preccurlyeq_p S'_i}{\lambda\langle +,+,X\rangle.\{F_x\}_{x\in X} : \&\{l_i{:}S_i\}_{i\in I} \preccurlyeq_p \&\{l_j{:}S'_j\}_{j\in J}}$$

$$[\oplus\cdot\oplus\text{ - }\preccurlyeq_p]\ \frac{\forall j \in J \subseteq I \quad F_j : S_j \preccurlyeq_p S'_j}{\lambda\langle +,+,X\rangle.\{F_j\}_{j\in J} : \bigoplus\{l_i{:}S_i\}_{i\in I} \preccurlyeq_p \bigoplus\{l_j{:}S'_j\}_{j\in J}}$$

$$[\boxplus\cdot\boxplus\text{ - }\preccurlyeq_p]\ \frac{\forall i \in I \subseteq J \quad F_i : S_i \preccurlyeq_p S'_i}{\lambda\langle \oplus,\oplus,X\rangle.\{F_x\}_{x\in X} : \boxplus\{l_i{:}S_i\}_{i\in I} \preccurlyeq_p \boxplus\{l_j{:}S'_j\}_{j\in J}}$$

$$[\oplus\cdot\boxplus\text{ - }\preccurlyeq_p]\ \frac{\forall k \in K \subseteq (I\cap J) \neq \emptyset \qquad F_k : S_k \preccurlyeq_p S'_k}{\lambda\langle +,\oplus,X\rangle.\{F_k\}_{k\in K} : \bigoplus\{l_i{:}S_i\}_{i\in I} \preccurlyeq_p \boxplus\{l_j{:}S'_j\}_{j\in J}}$$

where $p, q = +, -, \pm$.

**Fig. 6.** Witnessed subtyping relations.

**Definition 6.9** *(Witnessed subtyping).* The *witnessed subtyping* relations $\preccurlyeq_-, \preccurlyeq_+, \preccurlyeq_\pm \subseteq \mathsf{OrchF} \times \mathsf{OST} \times \mathsf{OST}$ (dubbed, respectively, *client-*, *server-* and *peer-subtyping*) are defined as follows:
$F : S \preccurlyeq_p S'$ ($p \in \{+,-,\pm\}$) whenever there is a derivation for $F : S \preccurlyeq_p S'$ in the system of Fig. 6.

It is now possible to prove that the witnessed subtyping relations are transitive via composition of functors.

**Lemma 6.10.** *Let $F : U \preccurlyeq_p V$ and $G : V \preccurlyeq_p W$, with $p \in \{+, -\pm\}$.*
*Then $F \,\mathbf{;}\, G$ is defined and*

$$F \,\mathbf{;}\, G : U \preccurlyeq_p W$$

**Proof.** We consider only the case $p = +$, the cases $p = -, \pm$ can be treated in a similar way. The proof is by simultaneous induction over the derivations of $F : U \preccurlyeq_+ V$ and $G : V \preccurlyeq_+ W$.

$$[\text{Ax - }\preccurlyeq_+]\ F = \lambda\varphi.\mathtt{1} : \mathsf{end} \preccurlyeq_+ V$$

By axiom $[\text{Ax - }\preccurlyeq_+]$ and definition of $\mathbf{;}$ , we get

$$[\text{Ax - }\preccurlyeq_+]\ F \,\mathbf{;}\, G = \lambda\varphi.\mathtt{1} : \mathsf{end} \preccurlyeq_+ W$$

$$[\text{S-Input - }\preccurlyeq_+]\ \frac{F_1 : U_1 \preccurlyeq_+ V_1 \quad F_2 : U_2 \preccurlyeq_+ V_2}{\lambda\langle\bullet^{\mathsf{L}}_{F_1}\rangle.F_2 : \mathbf{?}[U^+_1].U_2 \preccurlyeq_+ \mathbf{?}[V^+_1].V_2}$$

with $F = \lambda\langle\bullet^{\mathsf{L}}_{F_1}\rangle.F_2$, $U = \mathbf{?}[U^+_1].U_2$ and $V = \mathbf{?}[V^+_1].V_2$

By definition of $\preccurlyeq_+$ we have necessarily that $W = \mathbf{?}[W_1^+].W_2$ and

$$\frac{G_1 : V_1 \preccurlyeq_+ W_1 \quad G_2 : V_2 \preccurlyeq_+ W_2}{\lambda\langle \bullet^{\mathrm{L}}_{G_1}\rangle.G_2 : \mathbf{?}[V_1^+].V_2 \preccurlyeq_+ \mathbf{?}[W_1^+].W_2}\ [\text{S-Input} - \preccurlyeq_+]$$

with $G = \lambda\langle \bullet^{\mathrm{L}}_{G_1}\rangle.G_2$.
By induction, from $F_1 : U_1 \preccurlyeq_+ V_1$ and $G_1 : V_1 \preccurlyeq_+ W_1$ we know that $F_1 \,;\, G_1$ is defined and $F_1 \,;\, G_1 : U_1 \preccurlyeq_+ W_1$. Similarly, we have that $F_2 \,;\, G_2$ is defined and $F_2 \,;\, G_2 : U_2 \preccurlyeq_+ W_2$. We can now apply rule [S-Input - $\preccurlyeq$] obtaining

$$\lambda\langle \bullet^{\mathrm{L}}_{F_1 \,;\, G_1}\rangle.(F_2 \,;\, G_2) : \mathbf{?}[U_1^+].U_2 \preccurlyeq_+ \mathbf{?}[W_1^+].W_2$$

Since $F_1 \,;\, G_1$ and $F_2 \,;\, G_2$ are both defined, the thesis follows, since, by definition of orchestrator composition, we have $F \,;\, G = \lambda\langle \bullet^{\mathrm{L}}_{F_1 \,;\, G_1}\rangle.(F_2 \,;\, G_2)$.

$$\frac{\forall j \in J \subseteq I \quad F_j : U_j \preccurlyeq_+ V_j}{\lambda\langle +, +, X\rangle.\{F_j\}_{j\in J} : \bigoplus\{l_i{:}U_i\}_{i\in I} \preccurlyeq_+ \bigoplus\{l_j{:}V_j\}_{j\in J}}\ [\oplus\cdot\oplus - \preccurlyeq_+]$$

where $F = \lambda\langle +, +, X\rangle.\{F_j\}_{j\in J}$, $U = \bigoplus\{l_i{:}U_i\}_{i\in I}$ and $V = \bigoplus\{l_j{:}V_j\}_{j\in J}$.

By definition of $\preccurlyeq$ we have to consider two possible rules used in the conclusion of $G : V \preccurlyeq_+ W$, the first one being

$$\frac{\forall k \in K \subseteq (H \cap J) \neq \emptyset \qquad G_k : V_k \preccurlyeq_+ W_k}{\lambda\langle +, \oplus, X\rangle.\{G_k\}_{k\in K} : \bigoplus\{l_j{:}V_j\}_{j\in J} \preccurlyeq_+ \boxplus\{l_h{:}W_h\}_{h\in H}}\ [\oplus\cdot\boxplus - \preccurlyeq_+]$$

with $G = \lambda\langle +, \oplus, X\rangle.\{G_k\}_{k\in K}$.
By the induction hypothesis we get that, for any $k \in K \subseteq (H \cap J)$, $F_k \,;\, G_k$ is defined and $F_k \,;\, G_k : U_k \preccurlyeq_+ W_k$.
Now, since $K \subseteq I$, we can apply rule [$\oplus\cdot\oplus$-$\preccurlyeq_+$] in order to get, as needed,

$$F \,;\, G = \lambda\langle +, \oplus, X\rangle.\{F_k \,;\, G_k\}_{k\in K} : \bigoplus\{l_i{:}U_i\}_{i\in I} \preccurlyeq_+ \boxplus\{l_h{:}W_h\}_{h\in H}$$

The second possibility is

$$\frac{\forall k \in K \subseteq J \quad G_k : V_k \preccurlyeq_+ W_k}{\lambda\langle +, +, X\rangle.\{G_j\}_{j\in J} : \bigoplus\{l_j{:}V_j\}_{j\in J} \preccurlyeq_+ \bigoplus\{l_k{:}W_k\}_{k\in K}}\ [\oplus\cdot\oplus - \preccurlyeq_+]$$

with $G = \lambda\langle +, +, X\rangle.\{G_j\}_{j\in J}$. By the induction hypothesis we get that, for any $k \in K \subseteq J \subset I$, $F_k \,;\, G_k$ is defined and $F_k \,;\, G_k : U_k \preccurlyeq_+ W_k$.
So, we can apply rule [$\oplus\cdot\oplus$-$\preccurlyeq_+$] obtaining, as required

$$F \,;\, G = \lambda\langle +, +, X\rangle.\{F_k \,;\, G_k\}_{k\in K} : \bigoplus\{l_i{:}U_i\}_{i\in I} \preccurlyeq_+ \bigoplus\{l_k{:}W_k\}_{k\in K}$$

The remaining cases are similar. □

By means of the witnessed subtyping relations we are in place to define a more general notion of compliance. In particular for what concerns delegation, it is safe to send and receive channel-ends whose types are in subtype relation up to a functor.

**Definition 6.11** *(Compliance with witnessed subtyping, $\dashv^{\preccurlyeq}$).* The relation of *compliance with witnessed subtyping*, dubbed $\dashv^{\preccurlyeq}$, is defined by the axioms and rules of Fig. 7.

**Remark 6.12.** Comparing Definitions 6.11 and 2.3 we see that the only difference between the systems for $\dashv^{\preccurlyeq}$ and $\dashv$ is rule:

$$\frac{\mathsf{f} : S_2 \dashv^{\preccurlyeq} S_2' \qquad F : S_1 \preccurlyeq_p S_1'}{\bullet_F.\mathsf{f} : \mathbf{?}[S_1'^p].S_2 \dashv^{\preccurlyeq} \mathbf{!}[S_1^p].S_2'}\ [\text{Cmpl-}\mathbf{?}\cdot\mathbf{!}\text{S-}\preccurlyeq]$$

and its symmetric.

Indeed according to Definition 2.3 the only possibility to derive $\mathsf{f}' : \mathbf{?}[S_1'^p].S_2 \dashv \mathbf{!}[S_1^p].S_2'$ is that $\mathsf{f}' = \bullet.\mathsf{f}$ and $S_1^p = S_2^p$ (which is syntactic equality). On the other hand, since it is clearly the case that for all $S$ there is an $F_S$ such that $F_S : S \preccurlyeq_p S$, by identifying $\bullet.\mathsf{f}$ with $\bullet_{F_S}.\mathsf{f}$ we see that $\dashv \subseteq \dashv^{\preccurlyeq}$.

$$[\text{Cmpl-Ax-}\preccurlyeq]\quad \mathtt{1} : \mathsf{end} \dashv^{\preccurlyeq} S$$

$$[\text{Cmpl-!}\cdot\text{?val-}\preccurlyeq]\ \frac{\mathsf{f} : S \dashv^{\preccurlyeq} S'}{\bullet.\mathsf{f} : ![G].S \dashv^{\preccurlyeq} ?[G].S'} \qquad [\text{Cmpl-?}\cdot\text{!val-}\preccurlyeq]\ \frac{\mathsf{f} : S \dashv^{\preccurlyeq} S'}{\bullet.\mathsf{f} : ?[G].S \dashv^{\preccurlyeq} ![G].S'}$$

$$[\text{Cmpl-?}\cdot\text{!S-}\preccurlyeq]\ \frac{\mathsf{f} : S_2 \dashv^{\preccurlyeq} S_2' \qquad F : S_1 \preccurlyeq_p S_1'}{\bullet_F.\mathsf{f} : ?[S_1'^p].S_2 \dashv^{\preccurlyeq} ![S_1^p].S_2'} \qquad [\text{Cmpl-!}\cdot\text{?S-}\preccurlyeq]\ \frac{\mathsf{f} : S_2 \dashv^{\preccurlyeq} S_2' \qquad F : S_1 \preccurlyeq_p S_1'}{\bullet_F.\mathsf{f} : ![S_1^p].S_2 \dashv^{\preccurlyeq} ?[S_1'^p].S_2'}$$

$$[\text{Cmpl-}\oplus\cdot\&\text{-}\preccurlyeq]\ \frac{(\forall i \in I) \quad \mathsf{f}_i : S_i \dashv^{\preccurlyeq} S_i'}{\sum_{i\in I} l_i.\mathsf{f}_i : \bigoplus\{l_i{:}S_i\}_{i\in I} \dashv^{\preccurlyeq} \&\{l_j{:}S_j'\}_{j\in I\cup J}}$$

$$[\text{Cmpl-}\&\cdot\oplus\text{-}\preccurlyeq]\ \frac{(\forall i \in I) \quad \mathsf{f}_i : S_i \dashv^{\preccurlyeq} S_i'}{\sum_{i\in I} l_i.\mathsf{f}_i : \&\{l_j{:}S_j\}_{j\in I\cup J} \dashv^{\preccurlyeq} \bigoplus\{l_i{:}S_i'\}_{i\in I}}$$

$$[\text{Cmpl-}\boxplus\cdot\&\text{-}\preccurlyeq]\ \frac{\emptyset \neq H \subseteq I \cap J \quad (\forall h \in H)\ \mathsf{f}_h : S_h \dashv^{\preccurlyeq} S_h'}{\oplus_{h\in H} l_h.\mathsf{f}_h : \boxplus\{l_i{:}S_i\}_{i\in I} \dashv^{\preccurlyeq} \&\{l_j{:}S_j'\}_{j\in J}}$$

$$[\text{Cmpl-}\&\cdot\boxplus\text{-}\preccurlyeq]\ \frac{\emptyset \neq H \subseteq I \cap J \quad (\forall h \in H)\ \mathsf{f}_h : S_h \dashv^{\preccurlyeq} S_h'}{\oplus_{h\in H} l_h.\mathsf{f}_h : \&\{l_i{:}S_i\}_{i\in I} \dashv^{\preccurlyeq} \boxplus\{l_j{:}S_j'\}_{j\in J}}$$

where $p \in \{+, -, \pm\}$.

**Fig. 7.** Compliance with witnessed subtyping.

The main result of this section now shows that witnessed compliance behaves as expected w.r.t. the new definition of compliance, and hence w.r.t. the old one.

**Theorem 6.13.**

i) *If* $F : S_1 \preccurlyeq_+ S_2$ *and* $\mathsf{f} : S \dashv^{\preccurlyeq} S_1$*, then* $\mathsf{f}' : S \dashv^{\preccurlyeq} S_2$*, where* $\mathsf{f}' = \text{NF}(F(\mathsf{f}))$*;*
ii) *If* $F : S_1 \preccurlyeq_- S_2$ *and* $\mathsf{f} : S_1 \dashv^{\preccurlyeq} S$*, then* $\mathsf{f}' : S_2 \dashv^{\preccurlyeq} S$*, where* $\mathsf{f}' = \text{NF}(F(\mathsf{f}))$*.*

**Proof.** (i) We proceed by induction on the derivation of $\mathsf{f} : S \dashv^{\preccurlyeq} S_1$.
We show only the most relevant cases.

$[\text{Cmpl-Ax-}\preccurlyeq]$
$\mathtt{1} : \mathsf{end} \dashv^{\preccurlyeq} S_1$
where $S = \mathsf{end}$.

By definition of $\dashv^{\preccurlyeq}$, it also holds that $\mathtt{1} : \mathsf{end} \dashv^{\preccurlyeq} S_2$. It is now immediate to check that, by Definition 6.5, $\mathtt{1} = \text{NF}(F(\mathtt{1}))$.

$$[\text{Cmpl-?}\cdot\text{!S-}\preccurlyeq]\ \frac{\mathsf{g} : U_2 \dashv^{\preccurlyeq} U'_2 \qquad G : U_1 \preccurlyeq_+ U_1'}{\bullet_G.\mathsf{g} : ?[U_1'^{+}].U_2 \dashv^{\preccurlyeq} ![U_1^{+}].U_2'}$$

(The case of delegated channel-end of polarity '$-$' can be treated similarly).
By hypothesis, we have that $F : ![U_1^+].U_2' \preccurlyeq_+ S_2$. By definition of $\preccurlyeq_+$ we can infer that $S_2 = ![V_1^+].V_2$ and $F = \lambda\langle\bullet^{\mathrm{R}}_{F_1}\rangle.F_2$. where $F_1 : V_1 \preccurlyeq_+ U_1$ and $F_2 = U_2' \preccurlyeq_+ V_2$.

We have to prove that $\bullet_{F_1}\,;\,_G.\text{NF}(F_2(\mathsf{g})) : ?[U_1'^{+}].U_2 \dashv^{\preccurlyeq} ![V_1^+].V_2$,
since $\text{NF}((\lambda\langle\bullet^{\mathrm{R}}_{F_1}\rangle.F_2)(\bullet_G.\mathsf{g})) = \bullet_{F_1}\,;\,_G.\text{NF}(F_2(\mathsf{g}))$.

By the induction hypothesis, from $\mathsf{g} : U_2 \dashv^{\preccurlyeq} U'_2$ and $F_2 = U_2' \preccurlyeq_+ V_2$ we get that $\text{NF}(F_2(\mathsf{g})) : U_2 \dashv^{\preccurlyeq} V_2$.
Moreover, by Lemma 6.10, from $F_1 : V_1 \preccurlyeq_+ U_1$ and $G : U_1 \preccurlyeq_+ U_1'$, we get $F_1\,;\,G : V_1 \preccurlyeq_+ U_1'$.
So, by rule $[\text{Cmpl-?}\cdot\text{!S-}\preccurlyeq]$ we get $\bullet_{F_1\circ G}.\text{NF}(F_2(\mathsf{g})) : ?[U_1'^{+}].U_2 \dashv^{\preccurlyeq} ![V_1^+].V_2$.

[CMPL-&·⊕-≼]
$$\frac{(\forall i \in I) \quad \mathsf{g}_i : U_i \dashv^{\preccurlyeq} U'_i}{\sum_{i\in I} l_i.\mathsf{g}_i : \&\{l_j{:}U_j\}_{j\in I\cup J} \dashv^{\preccurlyeq} \bigoplus\{l_i{:}U'_i\}_{i\in I}}$$

By hypothesis, we have that $F : \bigoplus\{l_i{:}U'_i\}_{i\in I} \preccurlyeq_+ S_2$.
By definition of $\preccurlyeq_+$, it follows that $S_2$ can be of two different forms. We consider the two cases separately.

(1): $S_2 = \boxplus\{l_h{:}V_h\}_{h\in H}$
In such a case, $F : \bigoplus\{l_i{:}U'_i\}_{i\in I} \preccurlyeq_+ \boxplus\{l_h{:}V_h\}_{h\in H}$
where $F = \lambda\langle +, \oplus, X\rangle.\{F_k\}_{k\in K}$,
with $\emptyset \neq K \subseteq (I \cap H)$ and, for any $k \in K$, $F_k : U'_k \preccurlyeq_p V_k$.

We have to prove that $\oplus_{k\in K} l_k.\mathrm{NF}(F_k(\mathsf{g}_k)) : \&\{l_j{:}U_j\}_{j\in I\cup J} \dashv^{\preccurlyeq} \boxplus\{l_h{:}V_h\}_{h\in H}$,
since $\mathrm{NF}((\lambda\langle +, \oplus, X\rangle.\{F_k\}_{k\in K})(\sum_{i\in I} l_i.\mathsf{g}_i)) = \oplus_{k\in K} l_k.\mathrm{NF}(F_k(\mathsf{g}_k))$.

By the induction hypothesis, from $\mathsf{g}_i : U_i \dashv^{\preccurlyeq} U'_i$ for any $i \in I$ and from $F_k : U'_k \preccurlyeq_+ V_k$, for any $k \in K$,
with $K \subseteq ((I \cup J) \cap H)$, we get
$\mathrm{NF}(F_k(\mathsf{g}_k)) : U_k \dashv^{\preccurlyeq} V_k$. We can hence use rule [CMPL-&·⊞-≼] in order to
get $\oplus_{k\in K} l_k.\mathrm{NF}(F_k(\mathsf{g}_k)) : \&\{l_j{:}U_j\}_{j\in I\cup J} \dashv^{\preccurlyeq} \boxplus\{l_h{:}V_h\}_{h\in H}$, as needed.

(2): $S_2 = \bigoplus\{l_h{:}V_h\}_{h\in H}$
This case can be treated in a similar way as case (1).
(ii) Similar to item (i). □

Notice that in the previous theorem we have not taken into account the case when $F : S_1 \preccurlyeq_\pm S_2$. Such a case can actually be easily inferred by taking into account the following property.

**Proposition 6.14.** *$F : S_1 \preccurlyeq_\pm S_2$ iff $F : S_1 \preccurlyeq_+ S_2$ and $F : S_1 \preccurlyeq_- S_2$.*

**Proof.** By noticing that the rules of Fig. 6 are the same for all the subtyping relations, but the axioms. In particular, $F : S_1 \preccurlyeq_\pm S_2$ holds by [Ax - ≼±] iff it holds for both [Ax - ≼−] and [Ax - ≼+]. □

*6.2. Processes and their operational semantics with subtyping*

As mentioned before, the presence of subtyping implies a very simple extension of the set of processes, namely orchestrator-functors have to be present in the session-opening operators.

**Definition 6.15** *(Processes with subtyping).* We extend the set of processes of Definition 3.1, by replacing the session-opening operators with the following ones.

$$\begin{array}{rcll} P, Q & ::= & \vdots & \\ & | & \mathtt{request}_{S_F}(k)P & \text{session request} \\ & | & \mathtt{accept}_{S_F}(k)P & \text{session accept} \\ & & \vdots & \end{array}$$

where $F \in \mathsf{OrchF}$.

**Definition 6.16** *(Type system with subtyping, $\Vdash_{\preccurlyeq}$).* The type system with subtyping is obtained out of the system of Fig. 2, by simply replacing rules (Acc-T) and (REQ-T) with the following ones:

[Acc-T·≼]
$$\frac{\Gamma \Vdash_{\preccurlyeq} P\{k^+/k\} \triangleright \Delta \cdot k^+ : S' \qquad F : S \preccurlyeq_+ S'}{\Gamma \Vdash_{\preccurlyeq} \mathtt{accept}_{S_F}(k)P \triangleright \Delta}$$

[REQ-T·≼]
$$\frac{\Gamma \Vdash_{\preccurlyeq} P\{k^-/k\} \triangleright \Delta \cdot k^- : S' \qquad F : S \preccurlyeq_- S'}{\Gamma \Vdash_{\preccurlyeq} \mathtt{request}_{S_F}(k)P \triangleright \Delta}$$

**Remark 6.17.** Let us informally discuss the correctness of the above typing rules and the use of the functors labelling the session-opening operators. (We recall that, when not specified otherwise, we simply write $F(\mathsf{f})$ to denote $\mathrm{NF}(F(\mathsf{f}))$.)

Let us assume:

- to have a process of the form $\texttt{request}_{S_F}(k)Q$ which is well-typed since $Q$ is correctly typeable using $k^- : S_1$ with $F : S \preccurlyeq_- S_1$;
- to have a process $\texttt{accept}_{S'_{F'}}(k)Q'$ which is well-typed since $Q'$ is correctly typeable using $k^+ : S_2$ with $F' : S' \preccurlyeq_+ S_2$;
- that the session-accept and session-request operations of the above processes can be fired since $\mathsf{f} : S \dashv^{\preccurlyeq} S'$.

The interaction on the session opened by the above processes, in order it can safely proceed, cannot be orchestrated by the above f, but should be actually orchestrated by an f′ such that $\mathsf{f}' : S_1 \dashv^{\preccurlyeq} S_2$. Such an f′ is $F(F'(\mathsf{f}))$, as shown below.

By Theorem 6.13, from $\mathsf{f} : S \dashv^{\preccurlyeq} S'$ and $F' : S' \preccurlyeq_+ S_2$ we have that $F'(\mathsf{f}) : S \dashv^{\preccurlyeq} S_2$ and hence, by Theorem 6.13 again, from $F'(\mathsf{f}) : S \dashv^{\preccurlyeq} S_2$ and $F : S \preccurlyeq_- S_1$ we can get

$$F(F'(\mathsf{f})) : S_1 \dashv^{\preccurlyeq} S_2$$

Since, by Lemma 6.6, $(F' \,;\, F)(\mathsf{f}) = F(F'(\mathsf{f}))$, the above discussion intuitively justifies also the use of the functor $(F' \,;\, F)$ in rule [Link-$\preccurlyeq$] in Definition 6.18 below, where we modify the operational semantics of Fig. 3 by extending the reduction rules in which orchestrator-functors are needed.

**Definition 6.18** *(Reductions with subtyping).* We modify the operational semantics of Fig. 3 by replacing rules [Link] and [OrchDeleg] by the following ones:

[Link-$\preccurlyeq$]
$$\texttt{request}_{S_F}(k)P \mid \texttt{accept}_{S'_{F'}}(k)Q \longrightarrow (\nu k)(\,\langle k\rangle(F' \,;\, F)(\mathsf{f}) \;\mid\; P\{k^-/k\} \;\mid\; Q\{k^+/k\}\,) \qquad \text{if } \mathsf{f} : S \dashv^{\preccurlyeq} S'$$

[OrchDeleg-$\preccurlyeq$]
$$(\nu k)(\nu k')(\,\langle k\rangle \bullet_F .\mathsf{f} \mid \langle k'\rangle \mathsf{g} \mid \texttt{throw}\,k^p[k'^q].P \mid \texttt{catch}\,k^{\overline{p}}(k').Q \mid R) \longrightarrow (\nu k)(\nu k')(\,\langle k\rangle \mathsf{f} \mid \langle k'\rangle F(\mathsf{g}) \mid P \mid Q\{k'^q/k'\} \mid R) \qquad \text{if } F(\mathsf{g}) \in \mathsf{Orch}$$

The condition $F(\mathsf{g}) \in \mathsf{Orch}$ above guarantees the transformation performed by the functor does succeed.

Recall that, by our convention, $(F' \,;\, F)(\mathsf{f})$ and $F(\mathsf{g})$ actually denote their normal forms.

**Example 6.19** *(Extending the running example with subtyping).* Let us consider the types of our running Example 2.7, and modify them in order to take into account also subtyping.

We assume not to modify the client. So the type of the endpoint of the channel connecting it with a provider is still the following one.

$$\mathsf{ClntSess} = \mathbf{!}[\texttt{String}].\oplus\{\textsc{buy}{:}\oplus\{\textsc{uhd}{:}\mathsf{S}, \textsc{hd}{:}\mathsf{S}\},\ \textsc{rent}{:}\boxplus\{\textsc{uhd}{:}\mathsf{S}, \textsc{hd}{:}\mathsf{S}, \textsc{sd}{:}\mathsf{S}, \textsc{ld}{:}\mathsf{S}\}\}$$

where $\mathsf{S} = \mathbf{!}[\texttt{String}].\oplus\{\textsc{ok}{:}\mathbf{?}[\mathsf{PAY}^-].\mathbf{?}[\texttt{Url}], \textsc{no}{:}\mathsf{end}\}$ with

$$\mathsf{PAY} = \boxplus\{\textsc{Diners}{:}\mathbf{!}[\texttt{ccNumber}], \textsc{Mcard}{:}\mathbf{!}[\texttt{ccNumber}], \textsc{Visa}{:}\mathbf{!}[\texttt{ccNumber}]\}$$

We recall that, by the higher-order action type $\mathbf{?}[\mathsf{PAY}^-]$, the client is declaring to be able to receive a channel endpoint through which it can pay using a credit card among Diners, Mcard and Visa, if available.
We assume, instead, to have a different provider, such that the type of its endpoint be

$$\mathsf{ProvSess2} = \mathbf{?}[\texttt{String}].\&\{\textsc{buy}{:}\&\{\textsc{uhd}{:}\mathsf{S}', \textsc{hd}{:}\mathsf{S}'\},\ \textsc{rent}{:}\&\{\textsc{hd}{:}\mathbf{?}[\texttt{Nat}].\mathsf{S}', \textsc{sd}{:}\mathsf{S}', \textsc{ld}{:}\mathsf{S}'\}\}$$

where $\mathsf{S}' = \mathbf{?}[\texttt{String}].\oplus\{\textsc{ok}{:}\mathbf{!}[\mathsf{PAY2}^-].\mathbf{!}[\texttt{Url}], \textsc{no}{:}\mathsf{end}\}$ with

$$\mathsf{PAY2} = \oplus\{\textsc{Discover}{:}\mathbf{!}[\texttt{ccNumber}], \textsc{Mcard}{:}\mathbf{!}[\texttt{ccNumber}], \textsc{Visa}{:}\mathbf{!}[\texttt{ccNumber}]\}$$

By $\mathbf{!}[\mathsf{PAY2}^-]$, the provider is declaring that a channel endpoint will be sent, which can be used to freely choose among the credit cards Discover, Mcard and Visa. This is not precisely the sort of channel endpoint the client is waiting for. However, the behaviours PAY and PAY2 are compatible, that is they are in subtype relation, as we shall discuss shortly.

As in Example 2.7, the provider is assumed to establish a session with the bank. In the present modified example we assume the provider's channel endpoint to be typed by

$$\mathsf{bankCustSess2} = \mathbf{!}[\texttt{Amount}].\mathsf{PAY2}$$

The interaction between the client and the provider is guaranteed to be safe, since

$$\mathsf{g2} : \mathsf{ClntSess} \dashv^{\preccurlyeq} \mathsf{ProvSess2}$$

where $\mathsf{g2} = \bullet.(\,(\textsc{buy}.(\textsc{uhd}.\mathsf{g}' + \textsc{hd}.\mathsf{g2}')) + (\textsc{rent}.(\textsc{sd}.\mathsf{g2}' \oplus \textsc{ld}.\mathsf{g2}'))\,)$ with $\mathsf{g2}' = \bullet.(\,(\textsc{ok}.\bullet_F.\bullet) + \textsc{no}\,)$.

Notice that g2 differs from the g in Example 2.7 since the higher-order orchestration action is now labelled by the functor $F$, which enables to adapt an orchestrator for the delegated channel endpoint according to the differences between PAY and PAY2. In particular, we have that

$$F : \mathsf{PAY2} \preccurlyeq_{-} \mathsf{PAY}$$

where $F = \lambda\langle +, \oplus, X\rangle.\{G_k\}_{k\in\{\text{MCARD},\text{VISA}\}}$,
with $G_{\text{MCARD}} = G_{\text{VISA}} = \lambda\langle\bullet\rangle.\bullet.\lambda\varphi.1$
(For the sake of readability, we are identifying a label like $l_{\text{MCARD}}$ with MCARD)
We assume to have the very same bank of Example 2.7. We recall below the type describing its behaviour on the channel endpoint connected with the provider (and then with the client after the provider has delegated its channel endpoint to the client).

$$\mathsf{BankSess} = \mathbf{?}[\mathtt{Amount}].\&\{\text{DISCOVER}:\mathsf{bS}, \text{MCARD}:\mathsf{bS}, \text{VISA}:\mathsf{bS}, \text{AEXPR}:\mathsf{bS}\}$$

where $\mathsf{bS} = \mathbf{?}[\mathtt{ccNumber}].\mathbf{!}[\mathtt{TransIDnum}]$.
The correct interaction between the provider and the bank is guaranteed by the fact that

$$\mathsf{g3} : \mathsf{bankCustSess2} \dashv^{\preccurlyeq} \mathsf{BankSess}$$

where $\mathsf{g3} = \bullet.(\text{DISCOVER}.\bullet.1 + \text{MCARD}.\bullet.1 + \text{VISA}.\bullet.1)$
It is now not difficult to modify the process *Provider* of Example 4.3 in order to get a process *Provider2* which behaves according to the above types, and such that the evolution of system

$$\mathit{Client} \mid \mathit{Provider2} \mid \mathit{Bank}$$

is error-free. Error-freeness will be guaranteed by its typeability.
At the moment of the delegation, the orchestrator of the channel between *Provider2* and *Bank* will be

$$\text{DISCOVER}.\bullet.1 + \text{MCARD}.\bullet.1 + \text{VISA}.\bullet.1$$

By the reduction rule [OrchDeleg-$\preccurlyeq$], such a orchestrator will be replaced by

$$\text{MCARD}.\bullet.1 \oplus \text{VISA}.\bullet.1$$

since

$$\begin{array}{rl} & F(\mathsf{g3}) \\ = & \underline{(\lambda\langle +, \oplus, X\rangle.\{G_k\}_{k\in\{\text{MCARD},\text{VISA}\}})(\text{DISCOVER}.\bullet.1 + \text{MCARD}.\bullet.1 + \text{VISA}.\bullet.1)} \\ \longrightarrow & \text{MCARD}.(G_{\text{MCARD}}(\bullet.1)) \oplus \text{VISA}.(G_{\text{MCARD}}(\bullet.1)) \\ = & \text{MCARD}.(\underline{(\lambda\langle\bullet\rangle.\bullet.\lambda\varphi.1)(\bullet.1)}) \oplus \text{VISA}.((\lambda\langle\bullet\rangle.\bullet.\lambda\varphi.1)(\bullet.1)) \\ \longrightarrow & \text{MCARD}.\bullet.((\lambda\varphi.1)1) \oplus \text{VISA}.(\underline{(\lambda\langle\bullet\rangle.\bullet.\lambda\varphi.1)(\bullet.1)}) \\ \longrightarrow & \text{MCARD}.\bullet.(\underline{(\lambda\varphi.1)1}) \oplus \text{VISA}.\bullet.((\lambda\varphi.1)1) \\ \longrightarrow & \text{MCARD}.\bullet.1 \oplus \text{VISA}.\bullet.(\underline{(\lambda\varphi.1)1}) \\ \longrightarrow & \text{MCARD}.\bullet.1 \oplus \text{VISA}.\bullet.1 = \text{NF}(F(\mathsf{g3})) \end{array}$$

**Remark 6.20.** Notice that, in some settings, the flexibility provided by subtyping should be limited by particular policies in order to enforce safety. Let us see an example showing the need of restricting our use of subtyping during delegation.
Let us modify Example 6.19 above, by having the following type for the client endpoint of the *Client–Provider* session channel.

$$\mathsf{ClntSess2} = \mathbf{!}[\mathtt{String}].\bigoplus\{\text{BUY}:\bigoplus\{\text{UHD}:\mathsf{S}, \text{HD}:\mathsf{S}\},\ \text{RENT}:\boxplus\{\text{UHD}:\mathsf{S}, \text{HD}:\mathsf{S}, \text{SD}:\mathsf{S}, \text{LD}:\mathsf{S}\}\}$$

where $\mathsf{S} = \mathbf{!}[\mathtt{String}].\bigoplus\{\text{OK}:\mathbf{?}[\mathsf{end}^-].\mathbf{?}[\mathtt{Url}], \text{NO}:\mathsf{end}\}$.
Here the client does expect to receive a channel-end on which no interaction is possible ($\mathbf{?}[\mathsf{end}^-]$). Nonetheless, since trivially, for any $S$, we have that $S \preccurlyeq_{-} \mathsf{end}$, it follows that

$$\mathsf{g}'' : \mathsf{ClntSess2} \dashv^{\preccurlyeq} \mathsf{ProvSess2}$$

for a suitable $\mathsf{g}''$.

This means that we can type a system where the client, once the channel endpoint for the payment is received, does nothing on it.

Of course from the point of view of the *safeness* of the interaction, this is no problem at all: error freeness is what a type system has to guarantee. Nonetheless the above one is a behaviour we would like to limit. In cases like these we could hence restrict subtyping on delegated channel endpoints only to $\preccurlyeq_{\pm}$. By doing that, the Example 6.19 would still work, since

$$F : \mathsf{PAY2} \preccurlyeq_{\pm} \mathsf{PAY}$$

**Theorem 6.21** *(Subject reduction with subtyping). If* $\Gamma \Vdash_{\preccurlyeq} P \rhd \Delta$ *and* $P \longrightarrow Q$ *then* $\Gamma \Vdash_{\preccurlyeq} Q \rhd \Delta$.

**Proof.** By induction over the definition of $P \longrightarrow Q$. We take into account only the cases where functors are involved.

In case of rule [Link·$\preccurlyeq$] we have:

$$\mathtt{request}_{S_F}(k)P \mid \mathtt{accept}_{S'_{F'}}(k)Q \longrightarrow (\nu k)(\langle k\rangle(F'\,;F)(\mathsf{f}) \mid P\{k^-/k\} \mid Q\{k^+/k\})$$

where $\mathsf{f}$ is such that $\mathsf{f} : S \dashv S'$. On the other hand by hypothesis and the shape of the rules, we have the derivation:

$$\dfrac{\dfrac{\Gamma \Vdash_{\preccurlyeq} P\{k^-/k\} \rhd \Delta\cdot k^-{:}S_1 \quad F{:}S \preccurlyeq_{-} S_1}{\Gamma \Vdash_{\preccurlyeq} \mathtt{request}_{S_F}(k)P \rhd \Delta} \qquad \dfrac{\Gamma \Vdash_{\preccurlyeq} Q\{k^+/k\} \rhd \Delta\cdot k^+{:}S_2 \quad F'{:}S' \preccurlyeq_{+} S_2}{\Gamma \Vdash_{\preccurlyeq} \mathtt{accept}_{S'_{F'}}(k)Q \rhd \Delta'}}{\Gamma \Vdash_{\preccurlyeq} \mathtt{request}_{S_F}(k)P \mid \mathtt{accept}_{S'_{F'}}(k)Q \rhd \Delta\cdot\Delta'}$$

where $\Delta\cdot\Delta'$ has to be defined (namely $dom(\Delta) \cap dom(\Delta') = \emptyset$). This implies that:

$$(\Delta\cdot k^- : S_1)\cdot(\Delta'\cdot k^+ : S_2) = \Delta\cdot\Delta'\cdot\ k^- : S_1\ \cdot\ k^+ : S_2$$

is defined as well. By applying twice Theorem 6.13, as in Remark 6.17, we get $(F'\,;F)(\mathsf{f}) : S_1 \dashv S_2$. So, we can obtain the following derivation:

$$\dfrac{\dfrac{\Gamma \Vdash_{\preccurlyeq} P\{k^-/k\} \rhd \Delta\cdot k^- : S_1 \quad \Gamma \Vdash_{\preccurlyeq} Q\{k^+/k\} \rhd \Delta\cdot k^+ : S_2}{\Gamma \Vdash_{\preccurlyeq} P\{k^-/k\} \mid Q\{k^+/k\} \rhd \Delta\cdot\Delta'\cdot\ k^- : S_1\ \cdot\ k^+ : S_2}}{\Gamma \Vdash_{\preccurlyeq} (\nu k)(\langle k\rangle(F'\,;F)(\mathsf{f}) \mid P\{k^-/k\} \mid Q\{k^+/k\}) \rhd \Delta\cdot\Delta'}$$

In case of [OrchDeleg·$\preccurlyeq$] we have:

$$\begin{array}{l}(\nu k)(\nu k')(\ \langle k\rangle \bullet_F .\mathsf{f}\ \mid\ \langle k'\rangle \mathsf{g}\ \mid\ \mathtt{throw}\, k^p[k'^q].P\ \mid\ \mathtt{catch}\, k^{\overline{p}}(k').Q\ \mid\ R) \\ \longrightarrow\ (\nu k)(\nu k')(\ \langle k\rangle \mathsf{f}\ \mid\ \langle k'\rangle F(\mathsf{g})\ \mid P\ \mid\ Q\{k'^q/k'\}\ \mid\ R)\end{array}$$

Let us assume $p = +$ and $q = -$ (the other cases can be treated similarly).

Then by hypothesis there exists a derivation of the following form:

$$\dfrac{\dfrac{\dfrac{\mathcal{D}}{\Gamma \Vdash_{\preccurlyeq} (\nu k')(\ \langle k'\rangle \mathsf{g}\ \mid\ \mathtt{throw}\, k^+[k'^-].P\ \mid\ R) \rhd \Delta'\cdot\Delta_1\cdot k^+ : {!}[S'^-]S_2} \qquad \mathcal{D}'}{\Gamma \Vdash_{\preccurlyeq} (\nu k')(\langle k\rangle \bullet_F .\mathsf{f} \mid \langle k'\rangle \mathsf{g} \mid \mathtt{throw}\, k^+[k'^-].P \mid R) \mid \mathtt{catch}\, k^-(k').Q \rhd \widehat{\Delta}}}{\Gamma \Vdash_{\preccurlyeq} (\nu k)(\nu k')(\ \langle k\rangle \bullet_F .\mathsf{f}\ \mid\ \langle k'\rangle \mathsf{g}\ \mid\ \mathtt{throw}\, k^+[k'^-].P\ \mid\ \mathtt{catch}\, k^-(k').Q\ \mid\ R) \rhd \Delta}$$

where:

- ◇ $\widehat{\Delta} = \Delta'{\cdot}\Delta_1{\cdot}k^+{:}{!}[S'^-]S_2{\cdot}k^-{:}{?}[S''^-]S_1$
- ◇ $\mathcal{D} =$

$$\dfrac{\dfrac{\Gamma \Vdash_{\preccurlyeq} P \rhd \Delta'\cdot k^+{:}S_2}{\Gamma \Vdash_{\preccurlyeq} \mathtt{throw}\, k^+[k'^-].P \rhd \Delta'\cdot k'^-{:}S'\cdot k^+{:}{!}[S'^-]S_2} \qquad \Gamma \Vdash_{\preccurlyeq} R \rhd \Delta_1\cdot k'^+ : \widetilde{S''}}{\Gamma \Vdash_{\preccurlyeq} \mathtt{throw}\, k^+[k'^-].P\ \mid\ R \rhd \Delta'\cdot\Delta_1\cdot k^+ : {!}[S'^-]S_2\cdot k'^-{:}S'\cdot k'^+ : \widetilde{S''}}$$

- ◇ $\mathsf{g} : S' \dashv^{\preccurlyeq} \widetilde{S''}$.
- ◇ $\mathcal{D}' =$

$$\dfrac{\Gamma \Vdash_{\preccurlyeq} Q\{k'^-/k'\} \rhd \Delta''\cdot\ k'^-{:}S''\cdot\ k^-{:}S_1}{\Gamma \Vdash_{\preccurlyeq} \mathtt{catch}\, k^-(k').Q \rhd \Delta''\cdot\ k^-{:}{?}[S''^-]S_1}$$

- ◇ $\Delta'\cdot\Delta_1\cdot\Delta'' = \Delta$
- ◇ $\bullet_F.\mathsf{f} : {!}[S'^-]S_1 \dashv {?}[S''^-]S_2$.

We first notice that $\bullet_F.\mathsf{f} : \mathbf{!}[S'^-]S_1 \dashv \mathbf{?}[S''^-]S_2$ implies $F : S' \preccurlyeq_- S''$.

Now, by applying Theorem 6.13, from $\mathsf{g} : S' \dashv^{\preccurlyeq} \widetilde{S''}$ and $F : S' \preccurlyeq_- S''$ we can get

$$F(\mathsf{g}) : S'' \dashv^{\preccurlyeq} \widetilde{S''}$$

Hence it is possible to form the following derivation, since, from $\bullet_F.\mathsf{f} : \mathbf{!}[S'^-]S_1 \dashv \mathbf{?}[S''^-]S_2$ we have also that $\mathsf{f} : S_1 \dashv S_2$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \Vdash_{\preccurlyeq} Q\{k'^q/k'\} \rhd \Delta'' \cdot k'^-{:}S''\cdot k^-{:}S_1 \qquad \Gamma \Vdash_{\preccurlyeq} R \rhd \Delta_1\cdot k'^+ : \widetilde{S''}}{\Gamma \Vdash_{\preccurlyeq} Q\{k'^q/k'\} \mid R \rhd \Delta''\cdot\Delta_1\cdot k^-{:}S_1\cdot k'^-{:}S''\cdot k'^+ : \widetilde{S''}}}{\Gamma \Vdash (\nu k')(\langle k'\rangle F(\mathsf{g}) \mid Q\{k'^q/k'\} \mid R) \rhd \Delta_1\cdot\Delta''\cdot k^-{:}S_1} \qquad \Gamma \Vdash_{\preccurlyeq} P \rhd \Delta'\cdot k^+{:}S_2}{\Gamma \Vdash (\nu k')(\langle k\rangle \mathsf{f} \mid \langle k'\rangle F(\mathsf{g}) \mid P \mid\ Q\{k'^-/k'\}) \mid R \rhd \Delta\cdot k^-{:}S_1\cdot k^+{:}S_2}}{\Gamma \Vdash (\nu k)(\nu k')(\langle k\rangle \mathsf{f} \mid \langle k'\rangle F(\mathsf{g}) \mid P \mid\ Q\{k'^-/k'\} \mid R) \rhd \Delta} \qquad \square$$

Error freeness can be obtained as a simple corollary out of the subject reduction property, as for the system without subtyping.

## 7. Deterministic orchestrations and priorities

As recalled in the introduction and discussed in e.g. [8,11], in actual programming the only nondeterminism should be the one exhibited by the partners of an interaction. Notice that this cannot be achieved by simply taking into account only nondeterministic orchestrators. The nondeterminism would still remain inside the operational semantics since rule [Link] (or [Link·$\preccurlyeq$]) can nondeterministically *choose* among the possibly many deterministic orchestrators for a session. In the present section, in order to rule out nondeterminism from orchestration and from the operational semantics, we interpret speculative choices as priority lists, that is choices ordered by a preference order which can be represented either at type- or at process-level.

In order to focus on the relevant ideas concerning the use of priorities, subtyping will not be taken into account in the present section.

### 7.1. Type-level priorities

Our aim is that, when a speculative choice has to be performed, the orchestrator chooses the one with highest-priority among those which do not lead to a failure. Moreover, we wish the priorities to be represented inside the types.

Being the priorities totally ordered, the orchestrators will turn out to be deterministic in the following syntactic sense.

**Definition 7.1** *(Deterministic orchestrators).* An orchestrator $f \in \mathsf{Orch}$ is *deterministic* if it does not contain any occurrence of the $\oplus$ operator.

Deterministic orchestrators can be used to select exactly one safe option in a construct like $k \triangleleft [l_i{:}P_i]_{i\in I}$ according to a given priority ordering among the failure-amenable options $\{l_i\}_{i\in I}$. The priority ordering can be explicitly specified in the speculative types, which we interpret now as *speculative types with priorities*.

**Definition 7.2** *(Orchestrated session types with priorities).* We define the set OST-P of *Orchestrated Session Types with Priorities* by modifying the set OST of Definition 2.1 as follows:

we replace $\boxplus\{l_i{:}S_i\}_{i\in I}$ by $\boxplus\langle\langle l_1{:}S_1, \ldots, l_n{:}S_n\rangle\rangle$, with $n \geq 1$.

The option represented by the label $l_i$ $(1 \leq i \leq n-1)$ is assumed to have higher priority than the one represented by $l_{i+1}$.

A type of the form $\boxplus\langle\langle l_1{:}S_1, \ldots, l_n{:}S_n\rangle\rangle$ is called *prioritised speculative selection.*

In the present setting, the orchestrator for two compliant session types must be *priorities-aware*, in the following sense.

**Definition 7.3** *(Compliance with type-level priorities).* Let $\mathsf{f} \in \mathsf{Orch}$ and $S, S' \in \mathsf{OST\text{-}P}$.

We say $\mathsf{f}$ is *priorities-aware* for $S$ and $S'$ if $\mathsf{f} : S \dashv S'$ is derivable in the system of Fig. 1, where rules [Cmpl-$\boxplus$·&] and [Cmpl-&·$\boxplus$] are replaced, respectively, by

$$\frac{(\{1,\ldots,n\}\cap J)\neq\emptyset \qquad \mathsf{f}_c : S_c \dashv S'_c}{l_c.\mathsf{f}_c : \boxplus\langle\!\langle l_1{:}S_1,\ldots,l_n{:}S_n\rangle\!\rangle \dashv \&\{l_j{:}S'_j\}_{j\in J}}\ [\text{CMPL-}\boxplus\cdot\&\text{-PRTY}]$$

$$\frac{(\{1,\ldots,n\}\cap J)\neq\emptyset \qquad \mathsf{f}_c : S_c \dashv S'_c}{l_c.\mathsf{f}_c : \&\{l_j{:}S_j\}_{j\in J} \dashv \boxplus\langle\!\langle l_1{:}S'_1,\ldots,l_n{:}S'_n\rangle\!\rangle}\ [\text{CMPL-}\&\cdot\boxplus\text{-PRTY}]$$

where $c = \min\{h \mid h \in (\{1,\ldots,n\}\cap J)\ \&\ \mathsf{f}_h : S_h \dashv S'_h\}$ (the rules do not apply if the above minimum does not exist).
We alternatively say that $S$ and $S'$ are *priority compliant by* f.

It is immediate to check the following

**Fact 1.** *Let* f *be priorities-aware for* $S$ *and* $S'$*. Then* f *is deterministic and unique.*

Given $S, S \in \mathsf{OST\text{-}P}'$, the unique priorities-aware orchestrator for $S$ and $S'$, if any, can be computed by means of a simple proof-search in the system of Definition 7.3. Such a proof-search is formalized by the procedure **Synth** described in Fig. 8. It can be safely used in our formalism with type-level priorities since it is possible to prove the following correctness lemma.

**Lemma 7.4** *(**Synth** correctness).*

*i)* **Synth**$(S, S')$=**fail** *iff. There exists no priorities-aware orchestrator for* $S$ *and* $S'$*;*
*ii) Let* $\mathsf{f} = \mathbf{Synth}(S, S') \neq \mathbf{fail}$*. Then* $\mathsf{f} : S \dashv S'$*, where* f *is the unique deterministic priorities-aware orchestrator for* $S$ *and* $S'$*.*

**Proof.** The procedure **Synth** does correspond to a proof search in the system of Definition 7.3. In such a procedure, the proof search with arguments

$$\boxplus\{l_i{:}S_i\}_{i\in I} \dashv \&\{l_j{:}S'_j\}_{j\in J} \quad \text{or} \quad \&\{l_j{:}S_j\}_{j\in J} \dashv \boxplus\{l_i{:}S'_i\}_{i\in I}$$

is left to the auxiliary procedure $\mathbf{Synth^{fst}}$, which performs a proof search for each element of its argument list. $\mathbf{Synth^{fst}}$ stops as soon as one of such proof searches, if any, does succeed. This guarantees the returned orchestrator, if any, to always choosing the highest-priority option for *speculative types with priorities.* □

In our "prioritised setting" we can consider the same process calculus of Definition 3.1. For what concerns the type system, instead, we have to modify a rule in Fig. 2.

**Definition 7.5** *(Type system for* OST-P*).* The type system for orchestrated session types with priorities is the one of Fig. 2 where rule [CRes-T] is replaced by

$$\frac{\Gamma \Vdash P \triangleright \Delta\cdot k^- : S_1 \cdot k^+ : S_2 \qquad \mathsf{f} = \mathbf{Synth}(S_1, S_2) \neq \boldsymbol{\mathit{fail}}}{\Gamma \Vdash (\nu k)(\langle k\rangle \mathsf{f} \mid P) \triangleright \Delta}\ [\text{CRes-T-PRTY}]$$

Similarly, the **Synth** algorithm has to be taken into account also in the operational semantics.

**Definition 7.6** *(Operational semantics with type-level priorities).* The operational semantics for type-level priorities is the one of Fig. 3, but for rule [Link], which is now replaced by

[Link-Prty]
$$\mathtt{request}_S(k)P \mid \mathtt{accept}_{S'}(k)Q \longrightarrow (\nu k)(\langle k\rangle \mathsf{f} \mid P\{k^-/k\} \mid Q\{k^+/k\}) \quad \text{if } \mathsf{f} = \mathbf{Synth}(S, S') \neq \boldsymbol{\mathit{fail}}$$

The subject reduction property can be obtained by a simple adaptation of the proof of Theorem 4.10.

**Example 7.7.** Let us assume, as when priorities have been presented in the introduction, that the client of the of Example 2.7 has an order of preference among the following set of failure-amenable choices: {UHD, HD, SD, LD}. In particular, let us assume the option LD to be the one liked best and, in case that would not be available, SD to be the second preferred one, HD the third and UHD the last. In order to reflect such an order of preference, type ClntSess of Example 2.7 should be replaced by

$$\mathsf{ClntSessPr} = \mathbf{!}[\mathtt{String}].\bigoplus\{\text{BUY}:\bigoplus\{\text{UHD}:\mathsf{S}, \text{HD}:\mathsf{S}\},\ \text{RENT}:\boxplus\langle\langle \text{LD}:\mathsf{S}, \text{SD}:\mathsf{S}, \text{HD}:\mathsf{S}, \text{UHD}:\mathsf{S}\rangle\rangle\}$$

where $\mathsf{S} = \mathbf{!}[\mathtt{String}].\bigoplus\{\text{OK}:\mathbf{?}[\mathsf{PAY}^-].\mathbf{?}[\mathtt{Url}], \text{NO}:\mathsf{end}\}$

We can now apply the **Synth** procedure on ClntSessPr and ProvSess.

$$\mathbf{Synth}(\mathsf{ClntSessPr}, \mathsf{ProvSess}) = \widehat{\mathsf{g}}$$

where $\widehat{\mathsf{g}} = \bullet.(\,(\text{BUY}.(\text{UHD}.\mathsf{g}' + \text{HD}.\mathsf{g}')) + (\text{RENT}.\text{LD}.\mathsf{g}'))\,)$
with $\mathsf{g}' = \bullet.(\,(\text{OK}.\bullet.\bullet) + \text{NO}\,)$.

The definition of **Synth** does also guarantee an orchestrator always to respect the priorities described in speculative selection types. This however does not immediately guarantee that in any [OrchSSel]-redex the branch chosen by the orchestrator is the one with highest priority among those enabling a safe computation. In order to do that, we formalize the notion of *priority error*.

Informally, a priority error occurs whenever a "failure-free" computation would be possible also in case a label with higher priority were chosen in a [OrchSSel]-redex.

**Definition 7.8** *(Priority error). $P$ is a priority-error* if it contains a [OrchSSel]-redex

$$(\nu k)(\ \langle k\rangle l_c.\mathsf{f}\ \mid\ k^p \triangleleft [l_j : P_j]_{j\in\{1,..,n\}}\ \mid\ k^{\overline{p}} \triangleright \{l_i\!:\!Q_i\}_{i\in I}\ ) \qquad \text{with } c \in I \cap \{1,..,n\}$$

such that:

there exists $c' \in I \cap \{1,..,n\}$ and $\mathsf{f}' \in \mathsf{Orch}$ such that

1. $c' < c$ and
2. if $\widehat{P}$ is obtained out of $P$ by replacing the above redex with

$$(\nu k)(\ \langle k\rangle l_{c'}.\mathsf{f}'\ \mid\ k^p \triangleleft [l_j : P_j]_{j\in\{1,..,n\}}\ \mid\ k^{\overline{p}} \triangleright \{l_i\!:\!Q_i\}_{i\in I}\ ) \quad \text{with } c \in I \cap \{1,..,n\}$$

   the following properties hold:
   a) there exists a reduction sequence $\widehat{P} \longrightarrow^* Q$ such that $Q$ contains an irreducible potential $k$-redex (assuming, without loss of generality, all bound channel names to be distinct);
   b) For any reduction sequence $\widehat{P} \longrightarrow^* Q$ in which $Q$ contains an irreducible potential $k$-redex, then this has $\imath$ as orchestrator and end as process on the client's end of $k$.

**Remark 7.9.** Notice that the condition (2a) is necessary. Otherwise, a typable process like the following one would be a priority error.

$$(\nu k)(\langle k\rangle l_2.\imath \mid k^- \triangleleft [l_1\!:\!\mathtt{request}_{\mathbf{?}[\mathtt{Nat}]}(k^-![3]), l_2\!:\!\mathbf{0}] \mid k^+ \triangleright \{l_1\!:\!\mathtt{accept}_{\mathsf{end}}(k^+![5]), l_2\!:\!\mathbf{0}\})$$

since for $c' = 1$ and $\mathsf{f}' = \bullet.\bullet$ the condition (2b) would be vacuously satisfied.
(Notice that the request-accept pair could not be fired because $\mathbf{?}[\mathtt{Nat}] \not\dashv \mathsf{end}$)

**Remark 7.10.** In case of recursion be taken into account, we should modify the definition of priority-error by introducing the existence of an infinite reduction sequence containing an infinite number of $k$-redexes.

In order to extend error freeness to the absence of priority errors, we introduce an auxiliary notion of reduction and a lemma.

**Definition 7.11.**

i) Let $\mathsf{f} \in \mathsf{Orch}$ and $S, S' \in \mathsf{OST\text{-}P}$. Then $(\mathsf{f}, S, S')$ is called a priority-triple (triple for short).
ii) The reduction relation on priority-triples is defined as follows.
   - ◇ $(\bullet.\mathsf{f}, \mathbf{?}[T].S, \mathbf{!}[T].S') \longrightarrow (\mathsf{f}, S, S')$
   - ◇ $(\bullet.\mathsf{f}, \mathbf{!}[T].S, \mathbf{?}[T].S') \longrightarrow (\mathsf{f}, S, S')$
   - ◇ $(\sum_{i\in I} l_i.\mathsf{f}_i, \bigoplus\{l_i\!:\!S_i\}_{i\in I}, \&\{l_j\!:\!S'_j\}_{j\in I\cup J}) \longrightarrow (\mathsf{f}_k, S_k, S'_k) \qquad (k \in I)$
   - ◇ $(\sum_{i\in I} l_i.\mathsf{f}_i, \&\{l_j\!:\!S_j\}_{j\in I\cup J}, \bigoplus\{l_i\!:\!S'_i\}_{i\in I} \longrightarrow (\mathsf{f}_k, S_k, S'_k) \qquad (k \in I)$
   - ◇ $(l_c.\mathsf{f}_c, \&\{l_j\!:\!S_j\}_{j\in J}, \boxplus\langle\langle l_1\!:\!S'_1, \ldots, l_n\!:\!S'_n\rangle\rangle) \longrightarrow (\mathsf{f}_c, S_c, S'_c)$
   - ◇ $(l_c.\mathsf{f}_c, \boxplus\langle\langle l_1\!:\!S'_1, \ldots, l_n\!:\!S'_n\rangle\rangle, \&\{l_j\!:\!S_j\}_{j\in J}) \longrightarrow (\mathsf{f}_c, S_c, S'_c)$

**Lemma 7.12.** *Let $S = \boxplus\langle\langle l_1{:}S_1, \ldots, l_n{:}S_n\rangle\rangle$ and $S' = \&\{l_j{:}S'_j\}_{j\in J}$, and let $l_c.\mathsf{f}$ be a priorities-aware orchestrator for $S$ and $S'$.*
*Then, for any $c' < c$ and any $\mathsf{g} \in Orch$, there exists $\mathsf{g}'$, $V$ and $V'$ such that*

$$(l_c.\mathsf{g}, S, ![T].S') \longrightarrow^* (\mathsf{g}', V, V') \not\longrightarrow$$

*with $\mathsf{g}' \neq 1$ or $V \neq$ end.*

**Proof.** Easy, by Definitions 7.11 and 7.3. □

We are now ready to prove an error freeness property including absence of priority errors and guaranteeing the presence of only deterministic orchestrators.

**Proposition 7.13** *(Error freeness with priorities). If $P$ is a user-defined process such that $\Gamma \Vdash P \triangleright \Delta$ for some $\Gamma$ and $\Delta$, and $R$ is a run-time process such that $P \stackrel{*}{\longrightarrow} R$, then*

*i) $R$ is not an error;*
*ii) any orchestrator in $R$ is deterministic;*
*iii) $R$ is not a priority-error*

**Proof.** (i) Easy corollary of subject reduction.
(ii) By Lemma 7.4 and by the very definition of type system (in particular rule [CRes-T-Prty]) and of reduction rules (in particular the reduction [Link-Prty]), we are guaranteed the orchestrators used during a computation to be always deterministic.
(iii) We proceed by contradiction. Let us hence assume $R$ to be a priority-error and hence to have: a $k$-redex

$$(\nu k)(\ \langle k\rangle l_c.\mathsf{f} \ \mid\ k^p \triangleleft [l_j : P_j]_{j\in\{1,..,n\}} \ \mid\ k^{\overline{p}} \triangleright \{l_i{:}Q_i\}_{i\in I}\ ) \qquad \text{with } c \in I \cap \{1,..,n\}$$

and $c'$ and $\mathsf{f}'$, with the properties described in Definition 7.8. By typability of $R$, reasoning as in the proof of the subject reduction Theorem 4.10, we have that $\mathsf{f} = l_c.\mathsf{f}'$ is a priorities-aware orchestrator for $S = \boxplus\langle\langle l_1{:}S_1, \ldots, l_n{:}S_n\rangle\rangle$ and $S' = \&\{l_j{:}S'_j\}_{j\in J}$, which are, respectively the typings for $k^p$ and $k^{\overline{p}}$.

Let now $\widehat{R}$ be obtained, out of $R$, by replacing

$$(\nu k)(\ \langle k\rangle l_c.\mathsf{f} \ \mid\ k^p \triangleleft [l_j : P_j]_{j\in\{1,..,n\}} \ \mid\ k^{\overline{p}} \triangleright \{l_i{:}Q_i\}_{i\in I}\ ) \qquad \text{with } c' \in I \cap \{1,..,n\}$$

with

$$(\nu k)(\ \langle k\rangle l_{c'}.\mathsf{f}' \ \mid\ k^p \triangleleft [l_j : P_j]_{j\in\{1,..,n\}} \ \mid\ k^{\overline{p}} \triangleright \{l_i{:}Q_i\}_{i\in I}\ ) \qquad \text{with } c \in I \cap \{1,..,n\}$$

Let us now consider a reduction sequence $\widehat{P} \longrightarrow^* Q$ in which $Q$ contains an irreducible potential $k$-redex. Any $k$-reduction in the reduction sequence (but the possible clean-up ones) are driven by (the reducts of) $l_{c'}.\mathsf{f}'$ contained in the reduction sequence given by the application of Lemma 7.12. By definition of priority error, the irreducible potential $k$-redex should have $1$ as orchestrator and end as process on the client-end of $k$. But this would contradict Lemma 7.12. □

*7.2. Process-level priorities*

As mentioned in the introduction, it is possible to specify the priorities among failure-amenable options at process-level instead of at type-level. In order to do that, there is no need to change the set of types as provided in Definition 2.1. We change instead the speculative selection operator.

**Definition 7.14** *(Processes with explicit priorities).* We modify the set $P$ of processes in Definition 3.1 by replacing $k\triangleleft [l_i{:}P_i]_{i\in I}$ by $k\triangleleft\langle\langle l_1{:}P_1, \ldots, l_n{:}P_n\rangle\rangle$, with $n \geq 1$.
The option represented by the label $l_i$ $(1 \leq i \leq n-1)$ is assumed to have higher priority than the one represented by $l_{i+1}$.

By writing $k\triangleleft\langle\langle l_1{:}P_1, \ldots, l_n{:}P_n\rangle\rangle$ the programmer declares that she/he would like the computation to continue with $P_1$ if such a choice would not lead to a synchronization failure, or with $P_2$, if that were not the case, and so on.

In order the operational semantics can respect such intended meaning for the $k\triangleleft\langle\langle l_1{:}P_1, \ldots, l_n{:}P_n\rangle\rangle$ construct, one should know which are all the possible *safe choices* among $\{l_1, \ldots, l_n\}$. These can be all computed using the types and made available by the orchestrator.

So we define an algorithm $\mathbf{Synth^{UD}}$ such that $\mathbf{Synth^{UD}}(S, S')$ returns, if any, a (possibly nondeterministic) orchestrator exhibiting all safe options for speculative selection choices contained in $S$ or $S'$.

**Definition 7.15** *(The algorithm* $\textbf{Synth}^{\textbf{UD}}$*).*
The algorithm $\textbf{Synth}^{\textbf{UD}}$ is obtained out of the algorithm **Synth** of Fig. 8 by replacing the third ***else*** clause by the clause

$$\begin{array}{l} \textbf{else}\quad \textbf{if}\ (S = \&\{l_i{:}S_i\}_{i\in I}\ \textbf{and}\ S' = \boxplus\langle\!\langle l_j{:}S'_j\rangle\!\rangle_{j\in J}) \\ \qquad\qquad \textbf{or} \\ \qquad\quad (S = \boxplus\langle\!\langle l_j{:}S_j\rangle\!\rangle_{j\in J}\ \textbf{and}\ S' = \&\{l_i{:}S'_i\}_{i\in I}) \\ \qquad\quad \textbf{then let}\ \mathrm{res} = \textbf{Synth}^{\textbf{all}}\,([S_h \dashv S'_h]_{h\in I\cap J}) \\ \qquad\qquad\qquad \textbf{in if}\ \mathrm{res} \neq [\ \ ]\ \ \textbf{then}\ l_{c_1}.\mathsf{f}_1 \oplus \ldots \oplus l_{c_n}.\mathsf{f}_n\ \textbf{where}\ \mathrm{res} = [(\mathsf{f}_1, c_1), \ldots (\mathsf{f}_n, c_n)] \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{else fail} \end{array}$$

where $\textbf{Synth}^{\textbf{all}}$ is defined by

$$\begin{array}{l} \textbf{Synth}^{\textbf{all}}([\,]) = [\,] \\ \textbf{Synth}^{\textbf{all}}((S_c \dashv S'_c){:}\mathsf{xs}) = \\ \qquad \textbf{let}\ \mathsf{f} = \textbf{Synth}^{\textbf{UD}}(S_c \dashv S'_c)\ \textbf{in}\ \ (\textbf{if}\ \mathsf{f} =\ \textbf{fail then}\ \textbf{Synth}^{\textbf{all}}(\mathsf{xs})\ \textbf{else}\ (\mathsf{f}, c){:}\textbf{Synth}^{\textbf{all}}(\mathsf{xs})) \end{array}$$

The orchestrator, if any, returned by $\textbf{Synth}^{\textbf{UD}}(S, S')$ offer the maximum number of nondeterministic choices among all those witnessing $S \dashv S'$. So, in order to formalize such property, we provide the following definition.

**Definition 7.16.** Given $\mathsf{f} \in \mathsf{Orch}$, we define

$$\mathsf{complBy}(\mathsf{f}) \triangleq \{(S, S') \in \mathsf{Orch} \times \mathsf{Orch} \mid \mathsf{f} : S \dashv S'\}$$

The binary relation $\leq : \mathsf{Orch} \times \mathsf{Orch}$ is hence defined by

$$\mathsf{f} \leq \mathsf{g} \quad \triangleq \quad \mathsf{complBy}(\mathsf{f}) \subseteq \mathsf{complBy}(\mathsf{g})$$

When non-recursive orchestrators are used, we could provide an equivalent and purely syntactical definition of $\leq$. We use however the above one since it is abstract enough and can be used in presence of recursive orchestrators as well.

It is not difficult now to check the following property.

**Lemma 7.17.** *Let $H$ and $K$ be sets of indexes such that $H \subseteq K$. Then*

$$\oplus_{h\in H} l_h.\mathsf{f}_h \leq \oplus_{h\in K} l_h.\mathsf{f}_h$$

We can now prove that $\textbf{Synth}^{\textbf{UD}}$ corresponds to a proof search algorithm and that it returns the "most nondeterministic" orchestrator.

**Lemma 7.18** *($\textbf{Synth}^{\textbf{UD}}$ correctness).*

i) $\textbf{Synth}^{\textbf{UD}}(S, S') = \textbf{fail}$ *iff there exists no* $\mathsf{f}$ *such that* $\mathsf{f} : S \dashv S'$;
ii) *Let* $\mathsf{f} = \textbf{Synth}^{\textbf{UD}}(S, S') \neq \textbf{fail}$ *. Then* $\forall \mathsf{g}$ *s.t.* $\mathsf{g} : S \dashv S', \quad \mathsf{g} \leq \mathsf{f}$.

**Proof.** The algorithm $\textbf{Synth}^{\textbf{UD}}$ does implement a proof search in the system of Fig. 1. In the proof search, by means of the procedure $\textbf{Synth}^{\textbf{all}}$, all the orchestrators for the branches of speculative selection types are looked for. This means that, when the proof search deals with rules [Cmpl-⊞·&] and [Cmpl-&·⊞], the maximum set $H$ with the desired property is found. By inspection of the rules of Fig. 1, we can infer that any derivation for $S \dashv S'$ differ from the one for $\mathsf{f} : S \dashv S'$ by the cardinality of the sets $H$'s. Hence the thesis of (ii) descends by Lemma 7.17. □

**Remark 7.19.** The algorithms **Synth** and $\textbf{Synth}^{\textbf{UD}}$ can be adapted to the case of recursive orchestrators and types following the treatment of recursion in [11].

**Definition 7.20** *(Type system for process-level priorities).* The type system for process-level priorities is like the one of Fig. 2, but for rule [CRes-T], which is now replaced by the following one:

$$\frac{\Gamma \Vdash P \rhd \Delta \cdot k^- : S_1 \cdot k^+ : S_2 \qquad \mathsf{f} = \textbf{Synth}^{\textbf{UD}}(S, S') \neq \textbf{fail}}{\Gamma \Vdash (\nu k)(\langle k\rangle \mathsf{f} \mid P) \rhd \Delta}\ [\text{CRes-T-prtyP}]$$

$\mathbf{Synth}(S \dashv S')$ =

  **if** $S = \mathsf{end}$ **then** $\mathbb{1}$

  **else if** $(S = \mathbf{?}[G].S_1$ **and** $S' = \mathbf{!}[G].S'_1)$ **or** $(S = \mathbf{!}[G].S_1$ **and** $S' = \mathbf{?}[G].S'_1)$

    **then let** $\mathsf{f} = \mathbf{Synth}\,(S_1 \dashv S'_1)$

      **in if** $\mathsf{f} \neq$ **fail then** $\bullet.\mathsf{f}$ **else fail**

  **else if** $(S = \mathbf{?}[S_1^p].S_2$ **and** $S' = \mathbf{!}[S_1^p].S'_2)$ **or** $(S = \mathbf{!}[S_1^p].S_2$ **and** $S' = \mathbf{?}[S_1^p].S'_2)$

    **then let** $\mathsf{f} = \mathbf{Synth}(S_2 \dashv S'_2)$

      **in if** $\mathsf{f} \neq$ **fail then** $\bullet.\mathsf{f}$ **else fail**

  **else if** $(S = \&\{l_i{:}S_i\}_{i\in I}$ **and** $S' = \boxplus\langle\!\langle l_j{:}S'_j\rangle\!\rangle_{j\in J})$

    **or** $(S = \boxplus\langle\!\langle l_j{:}S_j\rangle\!\rangle_{j\in J}$ **and** $S' = \&\{l_i{:}S'_i\}_{i\in I})$

    **then let** $\mathsf{res} = \mathbf{Synth}^{\mathbf{fst}}\,([S_h \dashv S'_h]_{h\in I\cap J})$

      **in if** $(\mathsf{res} =$ **fail**$)$ **then fail else** $l_c.\mathsf{f}$ **where** $(\mathsf{f}, c) = \mathsf{res}$

  **else if** $(S = \bigoplus\{l_i{:}S_i\}_{i\in I}$ **and** $S' = \&\{l_j{:}S'_j\}_{j\in J})$

    **or** $(S = \&\{l_j{:}S_j\}_{j\in J}$ **and** $S' = \bigoplus\{l_i{:}S'_i\}_{i\in I})$

    **then let** $\forall i \in I.\mathsf{f}_i = \mathbf{Synth}(S_i, S'_i)$

      **in if** $\forall i \in I.\mathsf{f}_i \neq$ **fail then** $\sum_{i\in I} l_i.\mathsf{f}_i$ **else fail**

  **else fail**

 **where**

  $\mathbf{Synth}^{\mathbf{fst}}([\,])$ = **fail**

  $\mathbf{Synth}^{\mathbf{fst}}((S_c \dashv S'_c){:}\mathsf{xs})$ = **let** $\mathsf{f} = \mathbf{Synth}(S_c \dashv S'_c)$

      **in** (**if** $\mathsf{f} \neq$ **fail then** $(\mathsf{f}, c)$ **else** $\mathbf{Synth}^{\mathbf{fst}}(\mathsf{xs})$)

**Fig. 8.** The algorithm **Synth**.

**Remark 7.21.** It is worth remarking that the type system of Fig. 2 can remain as it is also for process-level priorities in case the subject reduction theorem be restricted to user-defined processes $P$ such that $\emptyset \Vdash P \triangleright \emptyset$.

**Definition 7.22** *(Operational semantics for process-level priorities).* The operational semantics for process-level priorities is obtained out of that of Figure by substituting rules [LinK] and [OrchSSel] with, respectively

[LinkPP]
$$\mathtt{request}_S(k)P \mid \mathtt{accept}_{S'}(k)Q \;\longrightarrow\; (\nu k)(\langle k\rangle \mathsf{f} \mid P\{k^-/k\} \mid Q\{k^+/k\}) \quad \text{if } \mathsf{f} = \mathbf{Synth}^{\mathbf{UD}}(S, S') \neq \mathbf{fail}$$

[OrchSSelPP]
$$(\nu k)(\langle k\rangle \textstyle\bigoplus_{i\in I}\mathsf{f}_i \mid k \triangleleft \langle\!\langle l_1{:}P_1, \ldots, l_n{:}P_n\rangle\!\rangle \mid k \triangleright \{l_j{:}Q_j\}_{j\in J}) \;\longrightarrow\; (\nu k)(\langle k\rangle \mathsf{f}_c \mid P_m \mid Q_c) \quad \text{where } c = \min(I \cap \{1, \ldots, n\})$$

Notice that, by means of the priority list, rule [OrchSSel] is deterministic even in presence of a nondeterministic orchestrator. So the only nondeterminism in a computation is the one due to the partners of an interaction.

The subject reduction theorem holds also for the present level-priorities setting. Its proof can be obtained out of the one for Theorem 4.10. Also in this case, error-freeness can be obtained as a simple corollary.

Notice that now, in a speculative-selection operation, the highest-priority label is guaranteed to the always chosen by the very definition of our operational semantics.

## 8. Conclusions and future work

We have defined a type system with binary session types for a calculus with orchestrated interactions. The condition for session opening is here more permissive than in usual calculi with session types in that the types of the processes participating in a session have just to be *compliant* rather than dual to each other. The relation of compliance stems from the theory of contracts [1–4]. In particular, our compliance relation has been inspired by the orchestrated compliance relations proposed in [8] and [11], where possible stuck states can be avoided by means of orchestrating processes. In the present paper the points where an orchestrator can affect a computation are made visible in the calculus by introducing a novel operator that we dub *speculative selection*. This implies extending the syntax of usual session types, so resulting in a session-type counterpart of the retractable session contracts of [9], where backtracking is taken into account instead of

orchestration. Typable processes are shown to be free from particular errors. An interpretation of speculative selection as an actual language construct is then provided in form of priority selection.

Subtyping has been introduced in our framework in a form resembling the use of explicit coercions. Subtyping judgements are equipped hence with orchestrator functors. They intervene during a computation by adapting those orchestrators that, due to the use of subtyping during the type-checking phase, do not reflect the structure of the current interaction.

Recursion has not been considered in the present paper, in order to focus on the main ideas and differences with other session-types formalisms. Adding recursion should not pose any technical difficulties and can be dealt with, for types and orchestrators, along the lines of [8] and [11]. Recursion for processes can be treated as done, among others, in [5].

An interesting extension of the present investigation is adapting speculative choices and orchestrated interactions in the setting of multiparty asynchronous sessions. A starting point would be to consider the *symmetric sum* operator investigated in [23]. Such an operator can be looked at as a sort of multiparty version of our speculative choice, in that it represents an agreement point between the participants of multiparty systems. In [23], however, there is no room for possible failures to be avoided, since the symmetric sum is used in global types representing the behaviour of failure-free systems.

In case one considered the multiparty session types of [24], it would be reasonable to associate an orchestrator to each channel involved in a multiparty session. The orchestrator would hence act as an input filter for the buffer associated to a channel. Of course the introduction of speculative selection in a multiparty scenario would imply to reconsider the relationship between global types and local types. Actually no speculative-selection type can reasonably come out by projecting a global type. So processes (and their related types) using speculative selections can be looked at as modules one can adapt, by means of orchestrators, in order to comply (precisely or at least) with the global interaction pattern represented by the global type. From a different perspective a global type, if any, could be synthesized out of a number of local types, similarly to what done in [25]. In our "speculative" setting, however, a global type should rather be considered as a global adaptor, a centralized orchestrator, something similar to the *medium process* of [26] whose action would now be restricted to driving the speculative choices.

By taking into account the discussion of Remark 2.6, it would be interesting to define a *complementary relation* as the composition of compliance and subtyping and investigating whether in our setting it can be interpreted as a generalization of duality.

If one wished to frame the present investigation in today's super-heterogeneous distributed-systems setting of microservices, it is worth noticing that behavioural types in general could result in a powerful tool to guarantee the compatibility of services. In particular, as discussed in [27], §4.

> "Behavioural interfaces are a hot topic right now and will likely play an important role in the future of microservices. We envision that they will also be useful for the development of automatic testing frameworks that check the communication behaviour of services."

Our relaxed notion of duality of session types goes in the direction of providing flexible and reliable notions of behavioural compatibility.

## Acknowledgements

We are very grateful to the referees for their careful reading and useful suggestions. We also thank Mariangiola Dezani-Ciancaglini for her support.

The authors were partially supported by the COST Action EUTYPES CA-15123 and, respectively, Project "Chance" of the University of Catania and Project FORMS 2015 of Turin.

## References

[1] C. Laneve, L. Padovani, The must preorder revisited: an algebraic theory for web services contracts, in: CONCUR'07, in: LNCS, vol. 4703, Springer, 2007, pp. 212–225.

[2] C. Laneve, L. Padovani, The pairing of contracts and session types, in: ICGT'08, in: LNCS, vol. 5065, Springer, 2008, pp. 681–700.

[3] G. Castagna, N. Gesbert, L. Padovani, A theory of contracts for web services, ACM Trans. Program. Lang. Syst. 31 (5) (2009) 19:1–19:61, https://doi.org/10.1145/1538917.1538920.

[4] G.T. Bernardi, M. Hennessy, Modelling session types using contracts, Math. Struct. Comput. Sci. 26 (3) (2016) 510–560, https://doi.org/10.1017/S0960129514000243.

[5] K. Honda, V. Vasconcelos, M. Kubo, Language primitives and type discipline for structured communication-based programing, in: ETAPS'98, in: LNCS, vol. 1381, 1998.

[6] N. Yoshida, V.T. Vasconcelos, Language primitives and type discipline for structured communication-based programming revisited: two systems for higher-order session communication, Electron. Notes Theor. Comput. Sci. 171 (4) (2007) 73–93, https://doi.org/10.1016/j.entcs.2007.02.056.

[7] F. Barbanera, U. de'Liguoro, Two notions of sub-behaviour for session-based client/server systems, in: PPDP, ACM Press, 2010, pp. 155–164.

[8] L. Padovani, Contract-based discovery of web services modulo simple orchestrators, Theor. Comput. Sci. 411 (2010) 3328–3347, https://doi.org/10.1016/j.tcs.2010.05.002.

[9] F. Barbanera, M. Dezani-Ciancaglini, I. Lanese, U. de'Liguoro, Retractable contracts, in: PLACES 2015, in: EPTCS, vol. 203, Open Publishing Association, 2016, pp. 61–72.

[10] F. Barbanera, U. de'Liguoro, A game interpretation of retractable contracts, in: Proceeding COORDINATION 2016, in: LNCS, vol. 9686, Springer, 2016, pp. 18–34.

[11] F. Barbanera, S. van Bakel, U. de'Liguoro, Orchestrated session compliance, J. Log. Algebraic Methods Program. 86 (1) (2017) 30–76, https://doi.org/10.1016/j.jlamp.2016.08.002.
[12] F. Barbanera, U. de'Liguoro, Sub-behaviour relations for session-based client/server systems, Math. Struct. Comput. Sci. 25 (6) (2015) 1339–1381, https://doi.org/10.1017/S096012951400005X.
[13] G. Bernardi, M. Hennessy, Using higher-order contracts to model session types, Log. Methods Comput. Sci. 12 (2) (2016), https://doi.org/10.2168/LMCS-12(2:10)2016.
[14] F. Barbanera, U. de'Liguoro, Session types for orchestrated interactions, in: Proceedings ICE'17, EPTCS, vol. 261, 2018, pp. 17–36, https://doi.org/10.4204/EPTCS.261.5.
[15] G. Bernardi, O. Dardha, S.J. Gay, D. Kouzapas, On duality relations for session types, in: TGC 2014, Springer, 2014, pp. 51–66.
[16] F. Barbanera, I. Lanese, U. de'Liguoro, Retractable and speculative contracts, in: COORDINATION'17, in: LNCS, vol. 10319, Springer, 2017.
[17] S. Gay, M. Hole, Subtyping for session types in the pi-calculus, Acta Inform. 42 (2/3) (2005) 191–225, https://doi.org/10.1007/s00236-005-0177-z.
[18] H. Hüttel, et al., Foundations of session types and behavioural contracts, ACM Comput. Surv. 49 (1) (2016) 3:1–3:36.
[19] B.C. Pierce, D. Sangiorgi, Typing and subtyping for mobile processes, Math. Struct. Comput. Sci. 6 (5) (1996) 409–453, https://doi.org/10.1017/S096012950007002X.
[20] V. Tannen, T. Coquand, C.A. Gunter, A. Scedrov, Inheritance as implicit coercion, Inf. Comput. 93 (1) (1991) 172–221, https://doi.org/10.1016/0890-5401(91)90055-7.
[21] A. Assaf, A calculus of constructions with explicit subtyping, in: 20th International Conference on Types for Proofs and Programs, TYPES 2014, May 12–15, 2014, Paris, France, 2014, pp. 27–46.
[22] F. Barbanera, U. de'Liguoro, Retractability, games and orchestrators for session contracts, Log. Methods Comput. Sci. 13 (3) (2017), https://doi.org/10.23638/LMCS-13(3:15)2017.
[23] L. Nielsen, N. Yoshida, K. Honda, Multiparty symmetric sum types, in: EXPRESS'10, in: EPTCS, vol. 41, 2010, pp. 121–135.
[24] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, J. ACM 63 (1) (2016) 9:1–9:67, https://doi.org/10.1145/2827695.
[25] J. Lange, A. Scalas, Choreography synthesis as contract agreement, in: ICE '13, in: EPTCS, vol. 131, Open Publishing Association, 2013, pp. 52–67.
[26] L. Caires, J.A. Pérez, Multiparty session types within a canonical binary theory, and beyond, in: FORTE 2016, Springer, 2016, pp. 74–95.
[27] N. Dragoni, S. Giallorenzo, A.L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina, Microservices: Yesterday, Today, and Tomorrow, Springer International Publishing, Cham, 2017, pp. 195–216.