Data-driven choreographies à la Klaim [⋆]

Roberto Bruni[1], Andrea Corradini[1], Fabio Gadducci[1], Hernán Melgratti[2], Ugo Montanari[1], and Emilio Tuosto[3]

[1] University of Pisa, Italy
[2] Universidad de Buenos Aires & Conicet, Argentina
[3] Gran Sasso Science Institute, Italy & University of Leicester, UK

Abstract. We propose Klaim as a suitable base for a novel choreographic framework. More precisely we advocate Klaim as a suitable language onto which to project *data-driven* global specifications based on distributed tuple spaces. These specifications, akin behavioural types, describe the coordination from a global point of view. Differently from behavioural types though, our specifications express the data flow across distributed tuple spaces rather than detailing the communication pattern of processes. We devise a typing system to validate Klaim programs against projections of our global specifications. An interesting feature of our typing approach is that well-typed systems have an arbitrary number of participants. In standard approaches based on behavioural types, this is often achieved at the cost of considerable technical complications.

1 Introduction

Communication-centered programming is playing a prominent role in the production of nowadays software. Programming peers that need to exchange information is an error-prone activity and the behaviour of even small systems is subject to a combinatorial blow-up as the number of peers increases. Therefore well-structured principles and rigorous foundations are needed to develop well-engineered, trustworthy software. One possibility is to exploit some sort of behavioural types [15,8] to manage abstract descriptions of peers and formally study their properties such as communication safety, absence of deadlocks, progress or session fidelity: given the types of the peers, the emerging behaviour of their composition is analysed. In the seminal paper [14], recently nominated the *most influential POPL paper (Award 2018)*, the authors push forward an abstract notion of global type of interaction that represents a sort of contract between the communicating peers. This is paired with the notion of local type that gives an abstract description of the behaviour of each peer, as taken in isolation. Interestingly, local types can be obtained "for free" by projection from

⋆ Research partly supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No 778233, by UBACyT projects 20020170100544BA and 20020170100086BA, and CONICET project PIP 11220130100148CO, by the EU COST Action IC1405, by the MIUR PRINs 201784YSZ5 *ASPRA: Analysis of program analyses* and 2017FTXR7S *IT-MaTTerS: Methods and tools for trustworthy smart systems*, and by University of Pisa PRA_2018_66 *DECLWARE: Metodologie dichiarative per la progettazione e il deployment di applicazioni.*

global types, while the properties of interest can be studied and guaranteed just at the level of global types, without the need of studying the composition of local types. The conformance of peers implementation w.r.t. the global type can be studied instead at the level of local types, allowing a more efficient form of type checking. Roughly this means that properties are stated globally but checked locally. Global types have been inspired by session types [13] and by choreography languages in service oriented computing (WS-CDL[4]), where complex interactions are modelled from the point of view of the global sequence of events that must take place in order to successfully complete the computation.

In the literature, global/local types have been studied mostly in the context of point-to-point channel-based interactions. This means that the main action in a choreography is the sending of a message from one peer to another on a specific channel (of a given type). In this paper we explore a different setting, where interaction over tuple-spaces replaces message passing, in the style of Linda-like languages [10]. Instead of primitives for sending and receiving messages, here there are primitives for inserting a tuple on a tuple space, for reading (without consuming) a tuple from a tuple space or for retrieving a tuple from a tuple space. We call these interactions data-driven, as decisions will be taken on the basis of the type of the tuples that are manipulated. We coined the term *klaimographies* in honour of the process language Klaim [6,1], a main contribution of Rocco De Nicola in the fields of process algebras and distributed programming. Inspired by Klaim, klaimographies exploit the notion of distributed tuple spaces to separate the access to data on the basis of the interactions that are carried out.

A marketplace scenario We illustrate this with a motivating example that we will formalise later on (cf. Example 5 on page 8). We consider a scenario where sellers and buyers use a marketplace provided by a broker. Sellers can put on sale (several) items and buyers can inspect them. When an item of interest is found, the client can start a negotiation with the seller. The intended behaviour of this choreography is informally represented by the BPMN diagram[5] in Fig. 1. The diagram does not specify the protocol in a precise way. In our scenario there is a single broker but an arbitrary number of sellers or buyers. This is not reflected in the diagram because the BMPN pools 'Seller' and 'Buyer' represent participants, not roles that maybe enacted by many participants. Taking into account multiplicity of participants triggers interesting issues. For instance, the bargaining subprocess should happen between two specific instances of participants: the buyer interested in a particular item and the seller that advertised such item. Moreover, the interactions among these specific instances must happen without interference from other participants.

There are several distinguishing features of klaimographies w.r.t. the literature on global types that tackle the issues described above. First, klaimographies naturally support an arbitrary number of participants. This is uncommon in standard behavioural types approaches where the number of participants in

[4] `http://www.w3.org/2002/ws/chor.`
[5] The diagram has been drawn with the BeePMN tool `https://www.beepmn.com`.

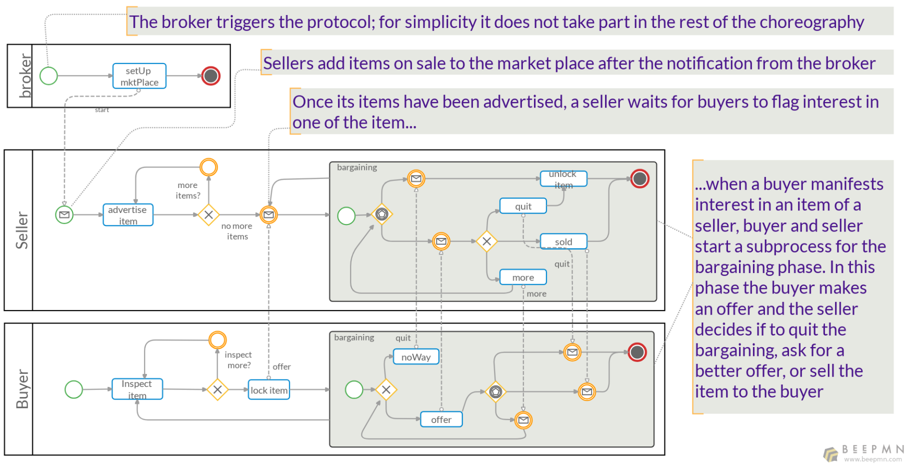
The broker triggers the protocol; for simplicity it does not take part in the rest of the choreography
Sellers add items on sale to the market place after the notification from the broker
Once its items have been advertised, a seller waits for buyers to flag interest in one of the item...
broker
setUp mktPlace
start
Seller
advertise item
more items?
no more items
bargaining
unlock item
quit
sold
quit
more
more
...when a buyer manifests interest in an item of a seller, buyer and seller start a subprocess for the bargaining phase. In this phase the buyer makes an offer and the seller decides if to quit the bargaining, ask for a better offer, or sell the item to the buyer
Buyer
Inspect item
inspect more?
offer
lock item
bargaining
quit
noWay
offer
BEEPMN
www.beepmn.com

Fig. 1. A marketplace scenario

interactions is usually fixed a priori, even when the number of participants is a parameter of the type, as done in [16] (see also Section 5). Second, interactions of klaimographies are multiway because each tuple can be read many times. Typically, session types specify point-to-point interactions where messages have exactly one producer and one consumer: see for instance [4] and the discussions on multiway interactions therein. Third, all interactions involve a tuple space locality instead of a channel name. Fourth, klaimographies are data-driven in the sense that they aim to check properties of data-flow. An example of use of klaimographies is to control the access to pieces of data in a tuple space.

The main contribution of this paper is to set up the formal setting of klaimographies and to prepare the ground for several interesting research directions: we fix the syntax of global and local types and define the projection from global to local types, as typical of choreographic frameworks. Global types are equipped with a partial order semantics of events and local types with an ordinary operational semantics. Then, the conditions under which the behaviour of projected local types is faithful to the semantics of global types are spelled out.

Shifting the focus from control to data in choreographic framework has several implications. Firstly, the emphasis is no longer on properties related to computational actors. For instance, klaimographies admit computations where some processes may not terminate and are left waiting for some data. In standard choreographic frameworks those would be undesired behaviours to rule out with suitable typing disciplines. Nonetheless, we claim that in some application domains computations with deadlocked processes have to be considered non-erroneous. For instance, in reactive systems based on event-notification frameworks some "listener" components must be kept waiting for events to occur. Our work paves the way to the formal study of properties of data, like consumption, persistence and availability, in a choreographic setting.

Another main innovation of klaimographies is that they allow one to easily represent protocols where a role can be enacted by an arbitrary number of components. We give an example of such protocol in Section 2.3. Remarkably, those protocols can be specified in some existing choreographic frameworks [16,5], but in a less abstract way that requires the explicit quantification on components.

Structure of the paper. After some preliminaries in Section 2.1, we define klaimographies as global types in Section 2.2 and give some examples in Section 2.3. In Section 3.1 we define the semantics of global types and give the adequacy conditions for projecting global types to local types. In Section 3.2 we define the syntax and operational semantics of local types and show how to project global types over local types in Section 3.3. The semantic correspondence between global types and local types is accounted for in Section 4. Some concluding remarks together with the discussion of related and future work are in Section 5.

2 Klaimographies

Our type system hinges on the basic notions of Klaim that are based on tuples, localities, and operations to generate and access tuple spaces. We recall that Klaim features two kinds of access to tuples located on a tuple space dubbed *input* and *read* access and often denoted as $\mathsf{in}\ t\,@\,\mathtt{l}$ and $\mathsf{read}\ t\,@\,\mathtt{l}$ in Klaim's literature. An input access $\mathsf{in}\ t\,@\,\mathtt{l}$ instantiates the variables in t corresponding to the fields in the matching tuple at locality $\mathtt{l}$ and then removes such tuple from $\mathtt{l}$, while a read access $\mathsf{read}\ t\,@\,\mathtt{l}$ does not remove the tuple from $\mathtt{l}$ after instantiating the variables in t. Section 2.1 introduces *tuple types* that basically abstract away from values in Klaim's tuples. Section 2.2 introduces *global types* meant to specify Klaim systems from a global point of view that, using *roles*, abstracts away from the actual instances of processes executing a protocol. Clearly, the form of interactions featured in the global types are inspired by Klaim operations.[6] Section 2.3 gives a taste of the expressiveness of our global types.

2.1 Tuple types

We consider a set of variables $\mathcal{V}$ ranged over by x and a set of localities $\mathcal{L}oc$ ranged over by $\mathtt{l}$ (and use ℓ to range over $\mathcal{L}oc \cup \mathcal{V}$) and we let $\mathtt{s}$ range over basic sorts which include `int`, `bool`, `str` and the sort `loc` of *localities*. The set $\mathcal{T}$ of *tuple (types)* consists of the terms derived from the following grammar:

$$\mathtt{t} ::= \mathtt{s} \;\big|\; \star \;\big|\; x:\mathtt{s} \;\big|\; \nu x:\mathtt{s} \;\big|\; \mathtt{t}\cdot\mathtt{t}$$

Tuple types are trees $\mathtt{t}\cdot\mathtt{t}$ where leaves are either a sort $\mathtt{s}$, any type $\star$, a sorted variable $x:\mathtt{s}$, or a fresh sorted variables $\nu x:\mathtt{s}$ (the difference between $x:\mathtt{s}$ and $\nu x:\mathtt{s}$ is clarified in Section 2.3). Note that $\nu x:\mathtt{s}$ are binders that *define* $x \in \mathcal{V}$.

[6] Klaim allows code mobility, which for the sake of simplicity is disregarded here. See however the discussion in Section 5

Hence, we talk about *free* and *defined* (sorted) names occurring in tuples. The functions $\mathit{fn}(\_)$ and $\mathit{dn}(\_)$ return sets of pairs $x \mapsto \mathtt{s}$ assigning sort $\mathtt{s}$ to $x \in \mathcal{V}$ and are given according to the definition below

$$\begin{array}{ll@{\qquad\qquad}ll}
\mathit{dn}(\mathtt{s}) & = \emptyset & \mathit{fn}(\mathtt{s}) & = \emptyset \\
\mathit{dn}(x : \mathtt{s}) & = \emptyset & \mathit{fn}(x : \mathtt{s}) & = \{x \mapsto \mathtt{s}\} \\
\mathit{dn}(\nu x : \mathtt{s}) & = \{x \mapsto \mathtt{s}\} & \mathit{fn}(\nu x : \mathtt{s}) & = \emptyset \\
\mathit{dn}(\mathtt{t}_1 \cdot \mathtt{t}_2) & = \mathit{dn}(\mathtt{t}_1) \cup \mathit{dn}(\mathtt{t}_2) & \mathit{fn}(\mathtt{t}_1 \cdot \mathtt{t}_2) & = \mathit{fn}(\mathtt{t}_1) \cup \mathit{fn}(\mathtt{t}_2) \\
\mathit{dn}(\star) & = \emptyset & \mathit{fn}(\star) & = \emptyset
\end{array}$$

We write $\llcorner\_\lrcorner$ to denote the projection of a set of pairs over its first component.

We say a tuple $\mathtt{t}$ is *well-sorted* if the following two conditions hold:

– $\llcorner \mathit{fn}(\mathtt{t}) \lrcorner \cap \llcorner \mathit{dn}(\mathtt{t}) \lrcorner = \emptyset$, i.e., free and defined names are disjoint; and
– $\mathtt{t} = \mathtt{t}_1 \cdot \mathtt{t}_2$ implies $\mathtt{t}_1$ and $\mathtt{t}_2$ are well-sorted and their names are disjoint, namely, $\llcorner \mathit{dn}(\mathtt{t}_1) \lrcorner \cap \llcorner \mathit{dn}(\mathtt{t}_2) \lrcorner = \emptyset$ and $\llcorner \mathit{fn}(\mathtt{t}_1) \lrcorner \cap \llcorner \mathit{fn}(\mathtt{t}_2) \lrcorner = \emptyset$.

Hereafter, we assume all tuples to be well-sorted. Note that $\mathit{fn}(\mathtt{t})$ and $\mathit{dn}(\mathtt{t})$ are partial functions (from names to sorts) for well-sorted tuples.

A substitution of the free occurrences of a variable x in a (well-sorted) tuple $\mathtt{t}$ by a variable $y \notin \mathit{dn}(\mathtt{t})$, written $\mathtt{t}\{^y/_x\}$, is defined by

$$(x : \mathtt{s})\{^y/_x\} = y : \mathtt{s} \quad \text{and} \quad (\mathtt{t}_1 \cdot \mathtt{t}_2)\{^y/_x\} = (\mathtt{t}_1\{^y/_x\}) \cdot (\mathtt{t}_2\{^y/_x\})$$

while it is the identity on the remaining cases. Let $\sigma = \{y_1/x_1, \ldots, y_n/x_n\}$ such that $x_i \neq x_j$ for all $i \neq j$ (i.e., σ is a partial endo-function on $\mathcal{V}$). We now write $\mathtt{t}\sigma$ for the simultaneous substitution of each x_i by y_i. We use Σ for the set of all substitutions. We write $\sigma_1\sigma_2$ for the composition of partial functions with disjoint domain, and $\sigma_1[\sigma_2]$ for the update of σ_1 with σ_2.

Tuple types $\mathtt{t}$ and $\mathtt{t}'$ such that $\mathit{dn}(\mathtt{t}) \cap \mathit{dn}(\mathtt{t}') = \emptyset$ can *match* by producing a substitution; this is realised by the partial function $\bowtie : \mathcal{T} \times \mathcal{T} \to \Sigma$ below

$$\mathtt{t} \bowtie \mathtt{t}' = \begin{cases} \emptyset & \text{if } \mathtt{t} = \star \vee \mathtt{t}' = \star \vee \mathtt{t}, \mathtt{t}' \in \{\mathtt{s}, x : \mathtt{s}\} \\ \sigma & \text{if } \mathtt{t} = \mathtt{t}_1 \cdot \mathtt{t}_2 \wedge \mathtt{t}' = \mathtt{t}'_1 \cdot \mathtt{t}'_2 \wedge \mathtt{t}_1 \bowtie \mathtt{t}'_1 = \sigma_1 \wedge \mathtt{t}_2\sigma_1 \bowtie \mathtt{t}'_2\sigma_1 = \sigma \\ \{^y/_x\} & \text{if } (\mathtt{t} = \nu y : \mathtt{s} \wedge \mathtt{t}' = x : \mathtt{s}) \vee (\mathtt{t}' = \nu y : \mathtt{s} \wedge \mathtt{t} = x : \mathtt{s}) \\ \mathit{undef} & \mathit{otherwise} \end{cases}$$

We write $\mathtt{t} \bowtie \mathtt{t}'$ when $\mathtt{t} \bowtie \mathtt{t}' = \sigma$ for a substitution $\sigma \in \Sigma$.

We say that $\mathtt{t}$ *generates* when in one of its fields there is a $\nu x : \mathtt{loc}$ type.

2.2 Global types

We fix two disjoint sets $\mathcal{U} = \{\mathsf{p}, \mathsf{q}, \ldots\}$ and $\mathcal{M} = \{\mathsf{P}, \mathsf{Q}, \ldots\}$, respectively of *unit* roles and *multiple* roles, and define the set of *roles* $\mathcal{R} = \mathcal{U} \cup \mathcal{M}$, ranged over by ρ. We conventionally write multiple roles with initial uppercase letters and unit roles with initial lowercase letters.

Roles have to be thought of as types inhabited by instances of processes enacting the behaviour specified in a choreography. Unit roles are unit types while multiple roles account for multiple instances of processes all performing actions according to their role.

Let us first define the grammar for *prefixes* used in global types:

$$\begin{array}{rcll}
\pi & ::= & \rho \,!\, (\mathtt{t}) \,@\, \ell & \text{(autonomous) output} \\
& \mid & \rho \,!\, \mathtt{t} \,@\, \ell & \text{(autonomous) read-only output} \\
& \mid & \rho \,?\, (\mathtt{t}) \,@\, \ell & \text{(autonomous) input} \\
& \mid & \rho \,?\, \mathtt{t} \,@\, \ell & \text{(autonomous) read} \\
& \mid & \rho \to \rho' : (\mathtt{t}) \,@\, \ell & \text{consuming interaction} \\
& \mid & \rho \to \rho' : \mathtt{t} \,@\, \ell & \text{read-only interaction}
\end{array}$$

We syntactically distinguish two kinds of prefixes. The prefixes generated by the first four productions in the grammar of π above are the *autonomous* prefixes, that is those prefixes that processes can execute directly on a tuple space without coordinating with other processes. They are analogous to Klaim primitives for Linda-like interactions. The prefixes generated by the remaining two productions are the *interaction* prefixes, namely those involving a role generating a tuple and one accessing it. They are analogous to the usual prefixes of global types. The set $\mathsf{roles}(\pi) \subseteq \mathcal{R}$ of roles in π is defined in the obvious way; note that $\mathsf{roles}(\pi)$ is a singleton if, and only if, π is an autonomous prefix. Inspired by Klaim, processes can access tuple types according to two modalities syntactically distinguished by the round brackets around the tuple in prefixes. More precisely, when a prefix surrounds a tuple $\mathtt{t}$ with round brackets then $\mathtt{t}$ is meant to be consumed, otherwise it is meant to be read-only.

We assume that tuple types used in read-only modalities do not generate.

Global types $\mathtt{K}$ have the following syntax

$$\mathtt{K} \;::=\; \sum_{i \in I} \pi_i.\mathtt{K}_i \;\;\big|\;\; \mathtt{K} \prec \mathtt{K} \;\;\big|\;\; X \;\;\big|\;\; \mu_\rho\; X.\mathtt{K}$$

where I is a finite set of indexes; we write $\mathbf{0}$ for $\sum_{i \in I} \pi_i.\mathtt{K}_i$ when $I = \emptyset$ (we omit trailing occurrences of $\mathbf{0}$) and $\pi_j.\mathtt{K}_j$ instead of $\sum_{i \in I} \pi_i.\mathtt{K}_i$ when $I = \{j\}$. The set $\mathsf{roles}(\mathtt{K}) \subseteq \mathcal{R}$ of roles of $\mathtt{K}$ is the set of roles that are mentioned in $\mathtt{K}$ and it is defined in the obvious way.

The syntax of global types features prefix guarded choices, sequential composition, and recursion. The semantics in Section 3.1 will make clear that sequential composition $\prec$ allows for some concurrency between actions in the absence of role and communication dependencies. To handle recursive behaviour, the construct $\mu_\rho\; X.\mathtt{K}$ singles out a role $\rho \in \mathsf{roles}(\mathtt{K})$ deciding whether to repeat the execution of the body $\mathtt{K}$ or (if ever) to end it. To achieve this, ρ notifies the decision to stop or to do a next iteration by generating tuple types for the other roles (this is formally defined in Section 3.1). We omit the decoration ρ when $\mathsf{roles}(\mathtt{K}) = \{\rho\}$.

We extend the notions of defined and free names to global types as follows:

$$fn(\rho\,!\,(\mathtt{t})\,@\,\ell) = fn(\mathtt{t}) \cup \{\ell \mapsto \mathtt{loc}\} \qquad dn(\rho\,!\,(\mathtt{t})\,@\,\ell) = dn(\mathtt{t})$$

(omitted prefixes are defined analogously)

$$\begin{array}{llll}
fn(\sum_{i\in I}\pi_i.\mathtt{K}_i) & = \bigcup_{i\in I} fn(\pi_i) \cup (fn(\mathtt{K}_i) \setminus dn(\pi_i)) & dn(\sum_{i\in I}\pi_i.\mathtt{K}_i) & = \bigcup_{i\in I} dn(\pi_i) \cup dn(\mathtt{K}_i) \\
fn(\mathtt{K}_1 \prec \mathtt{K}_2) & = fn(\mathtt{K}_1) \cup fn(\mathtt{K}_2) & dn(\mathtt{K}_1 \prec \mathtt{K}_2) & = dn(\mathtt{K}_1) \cup dn(\mathtt{K}_2) \\
fn(X) & = \emptyset & dn(X) & = \emptyset \\
fn(\mu_\rho\; X.\mathtt{K}) & = fn(\mathtt{K}) & dn(\mu_\rho\; X.\mathtt{K}) & = dn(\mathtt{K})
\end{array}$$

We remark that in $\mathtt{K}_1 \prec \mathtt{K}_2$ the scope of names defined in $\mathtt{K}_1$ does not include $\mathtt{K}_2$. We write $n(\_)$ for the set of (sorted) defined and free names of a term. A set S of sorted names is *consistent* if $x \mapsto \mathtt{s} \in S$ and $x \mapsto \mathtt{s}' \in S$ implies $\mathtt{s} = \mathtt{s}'$.

The sets of well-sorted prefixes and terms are defined inductively as follows:

– π is well-sorted if $fn(\pi) \cap dn(\pi) = \emptyset$ and $n(\pi)$ is consistent, i.e., there are no clashes/inconsistencies in the sorts of the names in the component $\mathtt{t}$ of π and the locality ℓ mentioned in π;
– $\mathtt{K} = \sum_{i\in I}\pi_i.\mathtt{K}_i$ is well-sorted if for all $i \in I$ both π_i and $\mathtt{K}_i$ are well-sorted and $n(\mathtt{K})$ is consistent;
– $\mathtt{K}_1 \prec \mathtt{K}_2$ is well-sorted if $\mathtt{K}_1$ and $\mathtt{K}_2$ are well-sorted and $n(\mathtt{K}_1 \prec \mathtt{K}_2)$ is consistent;
– X is well-sorted and $\mu_\rho\; X.\mathtt{K}$ is well-sorted if $\mathtt{K}$ is well-sorted.

We consider terms up-to α-renaming of defined names and recursion variables. Correspondingly, substitutions are capture avoiding, in the sense that defined names can be renamed to fresh names before any substitution is applied to a term. As usual we say that a global type $\mathtt{K}$ is *closed* when it does not contain free occurrences of recursion variables X or free occurrences of names.

2.3 Some examples

We give a few simple global types (Examples 1 to 4) to highlight some basic features of klaimographies as well as a more complex example (Example 5) to illustrate the kind of protocols our global types can capture.

Example 1. Consider the following global type that describes the interaction of a client $\mathtt{c}$ with a simple service $\mathtt{s}$ that converts integers into strings.

$$\mathtt{K}_{(1)} = \mathtt{c} \rightarrow \mathtt{s} : (\mathtt{int})\,@\,\mathtt{l}\,.\,\mathtt{s} \rightarrow \mathtt{c} : (\mathtt{str})\,@\,\mathtt{l}$$

The client $\mathtt{c}$ produces an integer value on the locality $\mathtt{l}$ meant to be consumed by the server $\mathtt{s}$, which in turn produces back the converted string for the client. ◇

Elaborating on the previous example we discuss a few features of our setting.

Example 2. Assume that we consider client and server in Example 1 as multiple instead of unit roles, and write

$$\mathtt{K}_{(2)} = \mathsf{C} \to \mathsf{S} : (\mathtt{int}) @ \mathtt{l} . \mathsf{S} \to \mathsf{C} : (\mathtt{str}) @ \mathtt{l}$$

In this case, $\mathtt{K}_{(2)}$ states that each integer produced by a client will be consumed by a server, which will in turn produce a string for one of the clients. ◇

The type in Example 2 does not ensure that clients consume the string conversion of the integer they produced, because all tuples are put at the same location $\mathtt{l}$. Name binders can be used to correlate tuples.

Example 3. Consider

$$\mathtt{K}_{(3)} = \mathsf{C} \to \mathsf{S} : (\nu x : \mathtt{int}) @ \mathtt{l} . \mathsf{S} \to \mathsf{C} : (x : \mathtt{int} \ \cdot \mathtt{str}) @ \mathtt{l}$$

The first interaction binds the occurrence of x in the second interaction. The use of x in the second interaction constraints the instances of S and C to share a tuple whose integer expression matches the integer shared in the first interaction. Despite the identifier is known only to the communicating instances, this does not forbid two clients to generate the same integer value. ◇

The klaimography in Example 3 does not establishes a one-to-one association between instances of C and S. In fact, an instance of C not necessarily interacts with the same instance of S in the two communications when two instances of C generate the same integer in the first interaction.

Example 4. A one-to-one correspondence can be achieved by using defined names for localities. Consider

$$\mathtt{K}_{(4)} = \mathsf{C} \to \mathsf{S} : (\mathtt{int} \cdot \nu x : \mathtt{loc}) @ \mathtt{l} . \mathsf{S} \to \mathsf{C} : (\mathtt{str}) @ x$$

As in Example 3, client and server instances establish a common fresh identity x in the first interaction; this time the identity is a locality meant to share tuples in subsequent communications: the second interaction can only take place between the two instances sharing x, because such locality is known only to them. ◇

The following example focuses on a more realistic scenario, allowing us to combine together most of the features of our framework. For readability, we use the notation $\mu^1_\rho \ X.\mathtt{K}$ for a recursive protocol where the body $\mathtt{K}$ is repeated at least once. Formally,[7]

$$\mu^1_\rho \ X.\mathtt{K} = \mathtt{K}\{^{\mu_\rho \ X.\mathtt{K}}/_X\}.$$

[7] The reader should not be confused by the meaning of $\mu_\rho \ X.\mathtt{K}$ being different from that of $\mathtt{K}\{^{\mu_\rho \ X.\mathtt{K}}/_X\}$: this is because iteration and termination require some implicit interactions driven by ρ towards the other roles in $\mathtt{K}$, as discussed in Section 3.1.

Example 5. The marketplace scenario described in Section 1 can be formalised by the following global type.

$$
\begin{array}{l}
\mathsf{broker} \to \mathsf{Seller} : \mathtt{start} \mathbin{@} \mathtt{m} . \\
\mu^1\ X.\mathsf{Seller}\,!\,(\mathtt{str}\cdot\mathtt{int}\cdot\nu l:\mathtt{loc}) \mathbin{@}\mathtt{m}\,.\,X \prec \\
\mu^1_{\mathsf{Buyer}}\ Y.\left(\begin{array}{l}
\mu\ Z.\mathsf{Buyer}\,?\,\mathtt{str}\cdot\mathtt{int}\cdot\mathtt{loc}\mathbin{@}\mathtt{m}\,.\,Z\prec\\
\mathsf{Buyer}\,?\,(i:\mathtt{str}\cdot p:\mathtt{int}\cdot\nu l:\mathtt{loc})\mathbin{@}\mathtt{m}\,.\\
\mu^1_{\mathsf{Seller}}\ W.\left(\begin{array}{l}
\mathsf{Buyer}\to\mathsf{Seller}:(i:\mathtt{str}\cdot o:\mathtt{int})\mathbin{@} l\,.\\
\quad \mathsf{Seller}\to\mathsf{Buyer}:(\mathtt{quit})\mathbin{@}l\,.\\
\quad \mathsf{Seller}\,!\,(i:\mathtt{str}\cdot p:\mathtt{int}\cdot\nu l:\mathtt{loc})\mathbin{@}\mathtt{m}\,.\\
\quad Y\\
\quad +\\
\quad \mathsf{Seller}\to\mathsf{Buyer}:(\mathtt{sold})\mathbin{@}l\,.\,Y\\
\quad +\\
\quad \mathsf{Seller}\to\mathsf{Buyer}:(\mathtt{more})\mathbin{@}l\,.\,W\\
+\\
\mathsf{Buyer}\to\mathsf{Seller}:(\mathtt{noway})\mathbin{@}l\,.\\
\quad \mathsf{Seller}\,!\,(i:\mathtt{str}\cdot p:\mathtt{int}\cdot\nu l:\mathtt{loc})\mathbin{@}\mathtt{m}\,.\\
\quad Y
\end{array}\right)
\end{array}\right)
\end{array}
$$

The broker is a unit role that triggers sellers to start advertising their items on the marketplace location $\mathtt{m}$. Sellers and buyers are modelled as multiple roles. Each seller advertises one or more items at $\mathtt{m}$ (see recursion at line 2). Each buyer can inspect the advertised items (line 3) and eventually start bargaining on a selected item of interest. Note that the consumption at line 4 instantiates a private location l between the instance of Seller advertising the item and the instance of Buyer interested in buying it. Location l is used to perform the bargaining phase. See Section 3.1 and Section 3.2 for the exact semantics.

The seller instance controls the recursion $\mu^1_{\mathsf{Seller}}\ W.\cdots$; the body of the recursive type lets the buyer sharing location l decide whether to stop the bargaining (by exchanging a `noway` tuple, in which case the seller re-advertises the unsold item at $\mathtt{m}$) or to make an offer to the seller (which can then decide either to stop the bargaining, or to struck a deal, or to ask for an higher offer). ◇

3 Semantics

We equip global types with a semantics based on pomsets, define projections from global to local types (that is abstractions of Klaim processes enacting the roles of global types), and define the operational semantics of local types.

3.1 Pomsets for klaimographies

We give semantics to global types using *partially-ordered multi-sets* (pomsets for short). Following [9], a *pomset* is an isomorphism class of labelled partially-ordered sets (lposet) where, fixed a set of labels $\mathcal{L}$, an lposet is a triple $(\mathcal{E}, \leq, \lambda)$, with $\mathcal{E}$ a set of events, $\leq$ is a partial order on $\mathcal{E}$, and $\lambda : \mathcal{E} \to \mathcal{L}$ a labelling function mapping events in $\mathcal{E}$ to labels in $\mathcal{L}$. Two lposets $(\mathcal{E}, \leq, \lambda)$ and $(\mathcal{E}', \leq', \lambda')$ are *isomorphic* if there is a bijection $\phi : \mathcal{E} \to \mathcal{E}'$ such that $e \leq e' \iff \phi(e) \leq' \phi(e')$ and $\lambda = \lambda' \circ \phi$. Intuitively, the partial order $\leq$ yields a causality relation among events; for $e \neq e'$, if $e \leq e'$ then e' is caused by e or, in other words, the occurrence of e' must be preceded by the one of e in any execution respecting the order $\leq$. Note that λ is not required to be injective: for $e \neq e' \in \mathcal{E}$, $\lambda(e) = \lambda(e')$ means that e and e' model different occurrences of the same action. In the following, $[\mathcal{E}, \leq, \lambda]$ denotes the isomorphism class of $(\mathcal{E}, \leq, \lambda)$, symbols $r, r', \dots$ (resp. $R, R', \dots$) range over (resp. sets of) pomsets, and we assume that pomset r contains at least one lposet which will possibly be referred to as $(\mathcal{E}_r, \leq_r, \lambda_r)$. The empty pomset is denoted as ϵ.

An event e is an *immediate predecessor* of an event e' (or equivalently e' is an *immediate successor* of e) in a pomset r if $e \neq e'$, $e \leq_r e'$, and for all $e'' \in \mathcal{E}_r$ such that $e \leq_r e'' \leq_r e'$ either $e = e''$ or $e' = e''$. We draw pomsets as (a variant[8] of) Hasse diagrams of the immediate predecessor relation; for instance, the pomset

$$[\{e_1, e_2, e_3, e_4, e_5\}, \{(e_1, e_2), (e_1, e_3), (e_1, e_4), (e_1, e_5), (e_4, e_5)\}, \lambda]$$

is more conveniently written as

$$\left[\begin{tikzcd} & e_1 \arrow[rr] \arrow[dl] \arrow[dr] & & e_4 \arrow[d] \\ e_2 & & e_3 & e_5 \end{tikzcd}\right]_\lambda \quad \text{or} \quad \left[\begin{tikzcd} & \lambda(e_1) \arrow[rr] \arrow[dl] \arrow[dr] & & \lambda(e_4) \arrow[d] \\ \lambda(e_2) & & \lambda(e_3) & \lambda(e_5) \end{tikzcd}\right]$$

In the definition of our semantics we follow a principle that distinguishes the nature of autonomous and interaction prefixes.

– A tuple type $\mathtt{t}$ generated by an autonomous output can be accessed by any instance of *any* other role. However, there is no obligation to access the tuple $\mathtt{t}$, hence our semantics has to contemplate the cases where no read or input of $\mathtt{t}$ happens.
– Interactions are slightly more subtle. Firstly, a tuple type $\mathtt{t}$ in a read-only interaction is meant to be eventually accessed by (an instance of) the receiving role. Secondly, the tuple type $\mathtt{t}$ of a consuming interaction must be eventually consumed by an instance of the receiving role. Thirdly, if $\mathtt{t}$ is in a consuming interaction, any instance of the receiving role is allowed to read $\mathtt{t}$ prior to its consumption.

To capture this semantics we label events with autonomous prefixes π, possibly decorated as ${}^{[i]}\pi$. Intuitively, e.g., a label ${}^{[i]}\rho\,?\,\mathtt{t} \mathbin{@} \ell$ (resp. ${}^{[i]}\rho\,!\,\mathtt{t} \mathbin{@} \ell$) represents the fact that the i^{th} instance of ρ reads (resp. produces) a tuple of type $\mathtt{t}$. Labels π not prefixed with $[\_]$ specify that the event can be performed by any instance of the role in π. Hereafter, we only deal with pomsets labelled as above.

[8] Edges of Hasse diagrams are usually not oriented; here we use arrows so to draw order relations between events also horizontally.

Also, we assign *basic pomsets* $\mathsf{bp}(i,\pi)$ to prefixes π. A basic pomset yields the causal relations of π imposed by the above design principle. For an autonomous prefix π we define $\mathsf{bp}(i,\pi) = \left\{ \left[\ {}^{[i]}\pi \ \right] \right\}$. For interaction prefixes we define

$$\mathsf{bp}(i, \rho \to \rho' : \mathtt{t} \mathbin{@} \ell) = \bigcup_{h \geq 1} \left\{ \left[\begin{tikzcd} & {}^{[i]}\rho\,!\,\rho' \cdot \mathtt{t} \mathbin{@} \ell \arrow[dl] \arrow[dr] & \\ e_1 \arrow[rr, phantom, "\cdots\cdots\cdots\cdots"] & & e_h \end{tikzcd} \right]_\lambda \right\}$$

$$\mathsf{bp}(i, \rho \to \rho' : (\mathtt{t}) \mathbin{@} \ell) = \bigcup_{h \geq 1} \left\{ \left[\begin{tikzcd} & {}^{[i]}\rho\,!\,(\rho' \cdot \mathtt{t}) \mathbin{@} \ell \arrow[dl] \arrow[dr] & \\ e_1 \arrow[dr] \arrow[rr, phantom, "\cdots\cdots\cdots\cdots"] & & e_h \arrow[dl] \\ & {}^{[i]}\rho'\,?\,(\rho' \cdot {\downarrow}\mathtt{t}) \mathbin{@} \ell & \end{tikzcd} \right]_\lambda \right\} \cup \left\{ \left[\begin{tikzcd} {}^{[i]}\rho\,!\,(\rho' \cdot \mathtt{t}') \mathbin{@} \ell \arrow[d] \\ {}^{[i]}\rho'\,?\,(\rho' \cdot {\downarrow}\mathtt{t}) \mathbin{@} \ell \end{tikzcd} \right] \right\}$$

where each read-only event e_j (with $1 \leq j \leq h$) is labelled as $\lambda(e_j) = \rho'\,?\,\rho' \cdot {\downarrow}\mathtt{t} \mathbin{@} \ell$ with ${\downarrow}\mathtt{t}$ the binder-free version of $\mathtt{t}$. Formally, ${\downarrow}\_$ is defined such that ${\downarrow}(\nu x : \mathtt{s}) = x : \mathtt{s}$, it is the identity on $\mathtt{s}$, $\star$ and $x : \mathtt{s}$ and it behaves homomorphically over $\_ \cdot \_$. Note that the tuples in the labels of the events are "prefixed" by the role ρ' meant to access them; this requires to extend $\mathtt{s}$ so to include $\mathcal{R}$.

We can now give the semantics of prefixes as follows

$$[\![\pi]\!] = \begin{cases} \mathsf{bp}(1,\pi) & \text{if } \pi \text{ autonomous } \wedge \mathsf{roles}(\pi) \subseteq \mathcal{U} \\ \bigcup_{i \geq 1} \mathsf{bp}(i,\pi) & \text{if } \pi \text{ autonomous } \wedge \mathsf{roles}(\pi) \not\subseteq \mathcal{U} \end{cases}$$

$$[\![\rho \to \rho' : \mathtt{t} \mathbin{@} \ell]\!] = \begin{cases} \mathsf{bp}(1, \rho \to \rho' : \mathtt{t} \mathbin{@} \ell) & \text{if } \rho \in \mathcal{U} \\ \bigcup_{i \geq 1} \mathsf{bp}(i, \rho \to \rho' : \mathtt{t} \mathbin{@} \ell) & \text{otherwise} \end{cases}$$

$$[\![\rho \to \rho' : (\mathtt{t}) \mathbin{@} \ell]\!] = \begin{cases} \mathsf{bp}(1, \rho \to \rho' : (\mathtt{t}) \mathbin{@} \ell) & \text{if } \rho \in \mathcal{U} \\ \bigcup_{i \geq 1} \mathsf{bp}(i, \rho \to \rho' : (\mathtt{t}) \mathbin{@} \ell) & \text{otherwise} \end{cases}$$

As customary in other choreographic approaches (see [15,8,12] and references therein), the semantics of (closed) global types considers only *well-formed* global types, namely those enjoying *well-sequencedness* and *well-branchedness*. With respect to standard notions, however, these concepts have some peculiarities which we now discuss.

The key points of well-sequencedness are highlighted in the following type

$$\rho_1 \to \rho_2 : (\mathtt{str} \cdot \star) \mathbin{@} \mathtt{l} \prec \rho_2 \to \rho_3 : (\mathtt{str} \cdot \mathtt{int}) \mathbin{@} \mathtt{l} \tag{1}$$

where an instance of ρ_2 transforms a pair generated by ρ_1 into a pair for ρ_3. The choreography (1) may be violated when ρ_1 generates a tuple of type $\mathtt{str} \cdot \mathtt{int}$. In fact, such a tuple could match the type consumed by ρ_3 and therefore ρ_3

could "steal" the tuple from ρ_2. The problem is due to the fact that the tuples are generated on the same location and they match each other. More generally, the problem arises when different interactions introduce races on tuple types. Formally, write $(\mathtt{t}, \mathtt{l}) \in \mathtt{K}$ when there is a prefix in $\mathtt{K}$ whose tuple type is $\mathtt{t}$ and whose location is $\mathtt{l}$; we say that $(\mathtt{t}, \mathtt{l})$ is *local* to $\mathtt{K}$ if either of the following holds:

- $\mathtt{K} = \sum_{i \in I} \pi_i.\mathtt{K}_i$ and there is $i \in I$ such that either $(\mathtt{t}, \mathtt{l})$ is local to $\mathtt{K}_i$ or π_i outputs $\mathtt{t}$ at $\mathtt{l}$ for consumption and there is $\mathtt{t}'$ in an input from $\mathtt{l}$ in $\mathtt{K}_i$ such that $\mathtt{t} \bowtie \mathtt{t}'$
- $\mathtt{K} = \mathtt{K}_1 \prec \mathtt{K}_2$ and either $(\mathtt{t}, \mathtt{l})$ is local to $\mathtt{K}_1$ or $(\mathtt{t}, \mathtt{l})$ is local to $\mathtt{K}_2$
- $\mathtt{K} = \mu_\rho\ X.\mathtt{K}'$ and $(\mathtt{t}, \mathtt{l})$ is local to $\mathtt{K}'$.

Our notion of well-sequencedness requires absence of races on tuple types: we say that $\mathtt{K}_1$ and $\mathtt{K}_2$ are *well-sequenced* ($ws(\mathtt{K}_1, \mathtt{K}_2)$ in symbols) if, for $i \neq j \in \{1, 2\}$,

- for all $(\mathtt{t}, \mathtt{l})$ local to $\mathtt{K}_i$ and for all $(\mathtt{t}', \mathtt{l}) \in \mathtt{K}_j$, $\mathtt{t} \bowtie \mathtt{t}'$ implies $(\mathtt{t}', \mathtt{l})$ is in a read-only prefix in $\mathtt{K}_j$
- for all $(\mathtt{t}, \mathtt{l})$ in an autonomous input prefix of $\mathtt{K}_i$ and for all $(\mathtt{t}', \mathtt{l})$ generated in $\mathtt{K}_j$, $\mathtt{t} \bowtie \mathtt{t}'$ implies $(\mathtt{t}', \mathtt{l})$ is in an autonomous output prefix in $\mathtt{K}_j$ for consumption.

Finally, the semantics of the sequential composition $\mathtt{K}_1 \prec \mathtt{K}_2$ is as follows:

$$[\![\mathtt{K}_1 \prec \mathtt{K}_2]\!] = \begin{cases} \{\mathsf{seq}([\![\mathtt{K}_1]\!], [\![\mathtt{K}_2]\!]) & \text{if } ws(\mathtt{K}_1, \mathtt{K}_2)\} \\ \textit{undef} & \text{otherwise} \end{cases}$$

where the auxiliary operation $\mathsf{seq}(\_, \_)$ sequentially composes pomsets r and r' so to make the actions of a role in r to precede its actions in r':

$$\mathsf{seq}(r, r') = [\mathcal{E}_r \cup \mathcal{E}_{r'}, \leq, \lambda_r \cup \lambda_{r'}]$$

where we assume that $\mathcal{E}_r \cap \mathcal{E}_{r'} = \emptyset$ and $\leq$ is the reflexive and transitive closure of $\leq_r \cup \leq_{r'} \cup \{(e, e') \in \mathcal{E}_r \times \mathcal{E}_{r'} \mid \mathsf{roles}(e) = \mathsf{roles}(e')\}$ (recall that the labels of events are autonomous prefixes for which roles is a singleton).

We now consider well-branchedness, the other condition of well-formedness. As usual [12], well-branchedness requires two conditions: single selector and knowledge of choices. This can be formalised by requiring that one process in the choice is *active*, namely it selects the branch to take, while the others are *passive*, namely they are informed of the chosen branch by inputting some information that unambiguously identifies each branch of the choice. We syntactically[9] enforce uniqueness of selectors; a choice with several branches, takes the form

$$\sum_{i \in I} \rho \rightarrow \rho_i : (\mathtt{t}_i) @ \ell_i.\mathtt{K}_i \tag{2}$$

[9] This is just for simplicity as we could adopt definitions similar to the ones in [11,12] at the cost of higher technical complexity.

namely the instance of ρ acts as *unique* selectors. Intuitively, a passive instance (for example one enacting role ρ_i) in (2) has to be able to ascertain which branch the selector decided when the choice was taken. A simple way to ensure this is to require that the first input actions of each passive role are pairwise "disjoint" (i.e. non matching tuples or different locations) among branches.

The conditions on active and passive processes alone are not enough: in our framework, the notion of well-branchedness is slightly complicated by the presence of multiple roles. For instance, even assuming unique selectors, many instances of a selector role could exercise choices concurrently. This may create confusion if different branches generate matching tuples on a locality as illustrated by next example.

Example 6. Let $\mathtt{K}_{\mathrm{bad}} = \mathsf{A} \rightarrow \mathsf{B} : (\mathtt{int}) @ \mathtt{1}.\mathtt{K}_1 + \mathsf{A} \rightarrow \mathsf{B} : (\mathtt{str}) @ \mathtt{1}.\mathtt{K}_2$ where

$$\mathtt{K}_1 = \mathsf{B} \rightarrow \mathsf{C} : (\mathtt{str}) @ \mathtt{1}.\mathsf{C} \rightarrow \mathsf{B} : (\mathtt{bool}) @ \mathtt{1} \quad \text{and} \quad \mathtt{K}_2 = \mathsf{B} \rightarrow \mathsf{C} : (\mathtt{bool}) @ \mathtt{1}$$

In $\mathtt{K}_{\mathrm{bad}}$ confusion may arise that may alter the intended data flow. In fact, if two groups of participants execute the choice taking different branches, the instance of C executing $\mathtt{K}_2$ in the second branch may receive the boolean that the instance of C in $\mathtt{K}_1$ executing the first branch generates for B. $\diamond$

Therefore we require that tuple types in different branches of a choice do not match when they are at the same locality and that if a branch of a choice involves a unit role then none of the branches of the choice involves multiple roles. This condition, dubbed *confusion-free branching* ensures that different "groups" of instances involved in concurrent resolutions of a choice do not "interfere" with each other. If a unit role is involved, only one group can resolve the choice. We remark that the above condition is not a limitation; in fact, we can pre-process branches of choices by adding an extra field in all tuples of the branch so to unequivocally identify on which branch the tuple type is used.

To sum up, a choice as in (2) is *well-branched*, written $wb(\{\bigcup_{i \in I} \pi_i.\mathtt{K}_i\})$, when it is confusion-free, there is a unique active role, all other roles are passive. So we define

$$[\![\sum_{i \in I} \pi_i.\mathtt{K}_i]\!] = \begin{cases} \{\epsilon\} & \text{if } I = \emptyset \\ \bigcup_{r \in [\![\pi_i]\!], r' \in [\![\mathtt{K}_i]\!]} \mathsf{seq}(r, r') & \text{if } wb(\{\bigcup_{i \in I} \pi_i.\mathtt{K}_i\}) \\ \mathit{undef} & \text{otherwise} \end{cases}$$

Finally, the semantic equation for $\mu_\rho\ X.\mathtt{K}$ requires some auxiliary functions:

$$\begin{aligned} \mathit{STOP}(\rho, \mathtt{K}, \widetilde{y}) &= \rho \rightarrow \rho_1 : (\mathtt{stop}) @ y_1 \prec \ldots \prec \rho \rightarrow \rho_n : (\mathtt{stop}) @ y_n \\ \mathit{LOOP}(\rho, \mathtt{K}, \widetilde{y}, \widetilde{y}') &= \rho \rightarrow \rho_1 : (\nu y_1' : \mathtt{loc}) @ y_1) \prec \ldots \prec \rho \rightarrow \rho_n : (\nu y_n' : \mathtt{loc}) @ y_n \end{aligned}$$

where $\mathsf{roles}(\mathtt{K}) = \{\rho, \rho_1, \ldots, \rho_n\}$ with $\rho \notin \{\rho_1, \ldots, \rho_n\}$ and $\widetilde{y} = y_1 \cdots y_n$ and $\widetilde{y}' = y_1' \cdots y_n'$. Then, we define

$$[\![\mu_\rho\ X.\mathtt{K}]\!] = \begin{cases} \bigcup_{h \geq 0} [\![\mathrm{unfold}_h(\mu_\rho\ X.\mathtt{K}, \mathit{fn}(\mathtt{K}), \widetilde{y}, \widetilde{y}')]\!] & \text{if } ws(\mathtt{K}\{\mathbf{0}/_X\}, \mathtt{K}\{\mathbf{0}/_X\}) \\ & \text{and } \widetilde{y} \cap \mathit{fn}(\mathtt{K}) = \emptyset \\ \mathit{undef} & \text{otherwise} \end{cases}$$

where

$$\text{unfold}_h(\mu_\rho\ X.\mathtt{K}, L, \widetilde{y}, \widetilde{y}') = \begin{cases} STOP(\rho, \widetilde{y}) & \text{if } h = 0 \\ LOOP(\rho, \mathtt{K}, \widetilde{y}, \widetilde{y}') \prec \mathtt{K}\{^{\mathtt{K}'}/_X\} & \text{otherwise} \end{cases}$$

where $\mathtt{K}' = \text{unfold}_{h-1}(\mu_\rho\ X.\mathtt{K}, L \cup \widetilde{y} \cup \widetilde{y}', \widetilde{y}', \widetilde{y}'')$ with $\widetilde{y}''$ fresh.

3.2 Local types

A *local type* $\mathtt{L}$, which describes the interaction from the perspective of a single role, is a term generated by the following grammar.

$$\begin{aligned} \kappa &::= \mathtt{t}\,!\,\ell \ \big|\ (\mathtt{t})\,?\,\ell \ \big|\ \mathtt{t}\,?\,\ell \\ \mathtt{L} &::= \sum_{i \in I} \kappa_i.\mathtt{L}_i \ \big|\ \mathtt{L} \fatsemi \mathtt{L} \ \big|\ \big(\mu X(\widetilde{x})\,.\,\mathtt{L}\big)\langle\widetilde{\ell}\rangle \ \big|\ X\langle\widetilde{\ell}\rangle \end{aligned}$$

Prefixes $\mathtt{t}\,!\,\ell$, $(\mathtt{t})\,?\,\ell$ and $\mathtt{t}\,?\,\ell$ respectively stand for the production, consumption and read of a tuple $\mathtt{t}$ at the locality ℓ. Differently from global types, local types do not distinguish the generation of read-only tuples from the ones that can be consumed. Also, we use the symbol $\fatsemi$ instead of $\prec$ to remark the fact that, on local types, the sequential operator $\fatsemi$ serialises all activities.

Formation rules for branching and sequential local types $\mathtt{L}$ are exactly the same as for global types; analogously we write $\mathbf{0}$ for an empty sum. The syntax of recursive local types deviates from global types to make explicit the localities used for coordinating the execution; consequently, process variables are parametric (the syntax for recursive types is borrowed from [2]). The term $\big(\mu X(\widetilde{x})\,.\,\mathtt{L}\big)\langle\widetilde{\ell}\rangle$ defines a process variable X with parameters $\widetilde{x}$ to be used in $\mathtt{L}$; the initial values of $\widetilde{x}$ are given by $\widetilde{\ell}$. Accordingly, the usage of a process variable is parameterised, i.e., $X\langle\widetilde{\ell}\rangle$. For any $\big(\mu X(\widetilde{x})\,.\,\mathtt{L}\big)\langle\widetilde{\ell}\rangle$, we assume that $|\widetilde{x}| = |\widetilde{\ell}|$ and $|\widetilde{x}| = |\widetilde{\ell'}|$ for any bound occurrence of $X\langle\ell'\rangle$ in $\mathtt{L}$.

The notions of free and defined names, well-sorted and closed terms are straightforwardly extended to local types; in $\big(\mu X(\widetilde{x})\,.\,\mathtt{L}\big)\langle\widetilde{\ell}\rangle$, X and $\widetilde{x}$ act as binders for the occurrence in $\mathtt{L}$. Substitution on local types is defined as follows.

$$\begin{aligned} (\mathtt{t}\,!\,\ell)\{^y/_x\} &= \mathtt{t}\{^y/_x\}\,!(\ell\{^y/_x\}) && \textit{if } x \notin dn(\mathtt{t}) \\ ((\mathtt{t})\,?\,\ell)\{^y/_x\} &= (\mathtt{t}\{^y/_x\})\,?(\ell\{^y/_x\}) && \textit{if } x \notin dn(\mathtt{t}) \\ (\mathtt{t}\,?\,\ell)\{^y/_x\} &= \mathtt{t}\{^y/_x\}\,?(\ell\{^y/_x\}) && \textit{if } x \notin dn(\mathtt{t}) \\ (\sum_{i \in I} \kappa_i.\mathtt{L}_i)\{^y/_x\} &= \sum_{i \in I} (\kappa_i\{^y/_x\}).(\mathtt{L}_i\{^y/_x\}) && \textit{if } \forall i.x \notin dn(\kappa_i) \\ (\mathtt{L}_1 \fatsemi \mathtt{L}_2)\{^y/_x\} &= \mathtt{L}_1\{^y/_x\} \fatsemi \mathtt{L}_2\{^y/_x\} \\ X\langle\widetilde{\ell}\rangle\{^y/_x\} &= X\langle\widetilde{\ell}\{^y/_x\}\rangle \\ (\big(\mu X(\widetilde{z})\,.\,\mathtt{L}\big)\langle\widetilde{\ell}\rangle)\{^y/_x\} &= \big(\mu X(\widetilde{z})\,.\,\mathtt{L}\{^y/_x\}\big)\langle\widetilde{\ell}\{^y/_x\}\rangle && \textit{if } \{x, y\} \cap \widetilde{z} = \emptyset \end{aligned}$$

As for global types, we consider terms up-to α-renaming.

We consider the following syntax for the run-time semantics of a set of local types running on a tuple space, dubbed *specification.*

$$\Delta ::= \emptyset \ \big|\ \Delta, \rho : \mathtt{L} \ \big|\ \Delta, \mathtt{t}\,@\,\mathtt{l}$$

[LOut]
$$\frac{dn(\mathtt{t})\ \mathit{fresh}}{\Delta, \rho : \mathtt{t}\,!\,\mathtt{l}.\mathtt{L} \xrightarrow{\rho:\downarrow\mathtt{t}\,!\,\mathtt{l}} \Delta, \rho : \mathtt{L}, \downarrow\mathtt{t} \,@\, \mathtt{l}}$$

[LIn]
$$\frac{\mathtt{t} \bowtie \mathtt{t}' = \sigma}{\Delta, \rho : (\mathtt{t})\,?\,\mathtt{l}.\mathtt{L}, \mathtt{t}' \,@\, \mathtt{l} \xrightarrow{\rho:(\mathtt{t}')\,?\,\mathtt{l}} \Delta, \rho : \mathtt{L}\sigma}$$

[LRd]
$$\frac{\mathtt{t} \bowtie \mathtt{t}' = \sigma}{\Delta, \rho : \mathtt{t}\,?\,\mathtt{l}.\mathtt{L}, \mathtt{t}' \,@\, \mathtt{l} \xrightarrow{\rho:\mathtt{t}'\,?\,\mathtt{l}} \Delta, \rho : \mathtt{L}\sigma, \mathtt{t}' \,@\, \mathtt{l}}$$

[LSum]
$$\frac{\Gamma, \rho : \kappa_j.\mathtt{L}_j \xrightarrow{\alpha} \Delta'}{\Delta, \Gamma, \rho : \sum_{i \in I} \kappa_i.\mathtt{L}_i \xrightarrow{\alpha} \Delta, \Delta'} \quad j \in I$$

[LSeq$_1$]
$$\frac{\Delta, \rho : \mathtt{L}_1 \xrightarrow{\alpha} \Delta', \rho : \mathtt{L}_1'}{\Delta, \rho : \mathtt{L}_1 \fatsemi \mathtt{L}_2 \xrightarrow{\alpha} \Delta', \rho : \mathtt{L}_1' \fatsemi \mathtt{L}_2}$$

[LSeq$_2$]
$$\frac{\Delta, \rho : \mathtt{L}_1 \xrightarrow{\alpha} \Delta', \rho : \mathbf{0}}{\Delta, \rho : \mathtt{L}_1 \fatsemi \mathtt{L}_2 \xrightarrow{\alpha} \Delta', \rho : \mathtt{L}_2}$$

[LRec]
$$\frac{\Delta, \rho : \mathtt{L}\{^{(\mu X(\widetilde{x})\,.\,\mathtt{L})}/_X\}\{^{\widetilde{\mathtt{l}}}/_{\widetilde{x}}\} \xrightarrow{\alpha} \Delta'}{\Delta, \rho : (\mu X(\widetilde{x})\,.\,\mathtt{L})\langle\widetilde{\mathtt{l}}\rangle \xrightarrow{\alpha} \Delta'}$$

Fig. 2. Semantics of local types

A specification is a multiset containing two kinds of pairs: $\rho : \mathtt{L}$ associates a role with a local type; while $\mathtt{t} \,@\, \mathtt{l}$ indicates that a tuple of type $\mathtt{t}$ is available at locality $\mathtt{l}$. We assume that when $\rho \in \mathcal{U}$ then there is at most one pair $\rho : \mathtt{L}$ in Δ. We write Γ to denote a specification containing only terms of the form $\mathtt{t} \,@\, \mathtt{l}$.

The definition of $fn(\_)$ is straightforwardly extended to specifications.

We give an operational semantics to local types defined inductively by the rules in Fig. 2, where labels α are of the form $\rho : \kappa$. Rule [LOut] accounts for the behaviour of a role ρ that generates a tuple type $\mathtt{t}$ at the locality $\mathtt{l}$. The operational semantics for the generation of a tuple $\mathtt{t}$ that contains binders ensures that each defined name is substituted by a fresh free variable (i.e., a variable that does not occur free in $\Delta, \rho : \mathtt{t}\,!\,\mathtt{l}.\mathtt{L}$). This is achieved by requiring (i) all bound names in $\mathtt{t}$ to be fresh by α-renaming them if necessary (i.e., $dn(\mathtt{t})$ fresh) and (ii) the generated tuple $\downarrow\mathtt{t}$ is the binder-free version of $\mathtt{t}$. Rule [LIn] handles the case in which a role ρ consumes a tuple specified as $\mathtt{t}$ from locality $\mathtt{l}$. In order for the consumption to take place, the requested tuple $\mathtt{t}$ should match a tuple $\mathtt{t}'$ available at the locality $\mathtt{l}$. Note that the substitution σ generated from the match is applied to the continuation $\mathtt{L}$ associated with the role ρ; the consumed tuple is eliminated from the locality $\mathtt{l}$. Rule [LRd] is analogous to [LIn], but the read tuple is not removed from the tuple space. Rule [LSum] accounts for a role that follows by choosing one of its enabled branches. The semantics of a recursive term $(\mu X(\widetilde{x})\,.\,\mathtt{L})\langle\widetilde{\mathtt{l}}\rangle$ is given by the rule [LRec], which unfolds the definition, i.e., $\mathtt{L}\{^{(\mu X(\widetilde{x})\,.\,\mathtt{L})}/_X\}$ and substitutes the formal parameters $\widetilde{x}$ of the recursive definition by the actual parameters $\widetilde{\mathtt{l}}$ via the substitution $\{^{\widetilde{\mathtt{l}}}/_{\widetilde{x}}\}$.

$$
\mathtt{K}\downharpoonright_\rho^\eta = \begin{cases}
\mathbf{0} & \text{if } \rho \notin \mathsf{roles}(\mathtt{K}) \\[1ex]
\mathtt{K}' \downharpoonright_\rho^\eta & \text{if } \mathtt{K} = \pi.\mathtt{K}' \text{ and } \rho \notin \mathsf{roles}(\pi) \\[1ex]
\mathtt{t}\,!\,\ell.(\mathtt{K}' \downharpoonright_\rho^\eta) & \text{if } \mathtt{K} = \rho\,!\,\mathtt{t}\,@\,\ell.\mathtt{K}' \quad \text{or } \mathtt{K} = \rho \to \rho' : \mathtt{t}\,@\,\ell.\mathtt{K}' \\
& \text{or } \mathtt{K} = \rho\,!\,(\mathtt{t})\,@\,\ell.\mathtt{K}' \quad \text{or } \mathtt{K} = \rho \to \rho' : (\mathtt{t})\,@\,\ell.\mathtt{K}' \\[1ex]
(\mathtt{t})\,?\,\ell.(\mathtt{K}' \downharpoonright_\rho^\eta) & \text{if } \mathtt{K} = \rho\,?\,(\mathtt{t})\,@\,\ell.\mathtt{K}' \quad \text{or } \mathtt{K} = \rho' \to \rho : (\mathtt{t})\,@\,\ell.\mathtt{K}' \\[1ex]
\mathtt{t}\,?\,\ell.(\mathtt{K}' \downharpoonright_\rho^\eta) & \text{if } \mathtt{K} = \rho\,?\,\mathtt{t}\,@\,\ell.\mathtt{K}' \quad \text{or } \mathtt{K} = \rho' \to \rho : \mathtt{t}\,@\,\ell.\mathtt{K}' \\[1ex]
\sum_{i\in I} (\pi_i.\mathtt{K}_i) \downharpoonright_\rho^\eta & \text{if } \mathtt{K} = \sum_{i\in I} \pi_i.\mathtt{K}_i \\
\mathtt{K}_1 \downharpoonright_\rho^\eta \,;\, \mathtt{K}_2 \downharpoonright_\rho^\eta & \text{if } \mathtt{K} = \mathtt{K}_1 \prec \mathtt{K}_2 \\[1ex]
\multicolumn{2}{l}{\big(\mu X(x)\,.(\mathtt{stop})\,?\,x.\mathbf{0} + ((\nu y : \mathtt{loc})\,?\,x.\mathtt{K}' \downharpoonright_\rho^{\eta, X\mapsto y})\big)\langle \phi\rho \rangle} \\
& \text{if } \mathtt{K} = \mu_{\rho'}^{\phi}\; X.\mathtt{K}',\ \rho \neq \rho',\ \text{and } \{x, y\} \cap (\mathit{fn}(\mathtt{K}') \cup \mathit{cod}(\eta)) = \emptyset \\[1ex]
\multicolumn{2}{l}{\big(\mu X(\widetilde{x})\,.\,\mathtt{stop}\,!\,x_1 \dots \mathtt{stop}\,!\,x_n.\mathbf{0} + \nu y_1 : \mathtt{loc}\,!\,x \dots \nu y_n : \mathtt{loc}\,!\,x.\mathtt{K}' \downharpoonright_\rho^{\eta, X\mapsto \widetilde{y}}\big)\langle \phi\rho_1 \dots \phi\rho_n \rangle} \\
& \text{if } \mathtt{K} = \mu_{\rho}^{\phi}\; X.\mathtt{K}', \mathit{dom}(\phi) = \{\rho_1, \dots, \rho_n\}, \widetilde{x} = x_1 \dots x_n, \\
& \widetilde{y} = y_1 \dots y_n,\ \text{and } (\widetilde{x} \cup \widetilde{y}) \cap (\mathit{fn}(\mathtt{K}') \cup \mathit{cod}(\eta)) = \emptyset \\[1ex]
X\langle \eta X \rangle & \text{if } \mathtt{K} = X
\end{cases}
$$

Fig. 3. Projection

3.3 Obtaining local types out of global types

The projection of a global type $\mathtt{K}$ over a role ρ, written $\mathtt{K}\downharpoonright_\rho$, denotes the local type that specifies the behaviour of ρ in $\mathtt{K}$. Our projection operation is fairly standard but for the case of recursive types, which coordinate their execution by communicating over dedicated locations. Note that the semantics of recursive global types $\mu_\rho\; X.\mathtt{K}$ introduces auxiliary interactions to coordinate their execution (see $\mathit{STOP}(\rho, \mathtt{K}, \widetilde{y})$ and $\mathit{LOOP}(\rho, \mathtt{K}, \widetilde{y}, \widetilde{y}')$ in Section 3.1). However, there is not such an implicit mechanism in the execution of local types, where recursion is standard. Consequently, those auxiliary interactions need to be defined explicitly in local types; and consequently, they are introduced by projection (similarly to the approach in [3]). Another subtle aspect of the semantics of a recursive global type is that each iteration is parametric with respect to the set of localities used for coordination. In fact, $\mathit{LOOP}(\rho, \mathtt{K}, \widetilde{y}, \widetilde{y}')$ generates a set of fresh localities that are used by the next iteration. Such behaviour is mimicked by local types by relying on parameterised process variables. As a consequence, projection depends on the locations that are chosen as parameters of process variables. Hence, $\mathtt{K}\downharpoonright_\rho$ is defined in terms of $\mathtt{K}\downharpoonright_\rho^\eta$, where η is a partial function that maps process variables into sequences of locations, i.e., $\eta X = \widetilde{\ell}$; and $\mathtt{K}\downharpoonright_\rho = \mathtt{K}\downharpoonright_\rho^\emptyset$. We now comment on the definition of $\mathtt{K}\downharpoonright_\rho^\eta$ in Fig. 3. As usual, the local type corresponding to a role ρ that is not part of $\mathtt{K}$ is $\mathbf{0}$. The projection of a prefix π depends on the role played by ρ in π: it is omitted when ρ does not participate on π; it is the production of a tuple when π is an interaction or an autonomous output and ρ is the producer; it is the consumption of a tuple when π is an autonomous input or a consuming interaction and ρ is the consumer; or else it is the read of a tuple. Projection is homomorphic with respect to choices and sequential composition.

A global type $\mu_\rho\ X.\mathtt{K}$ is projected as a recursive local type $\big(\mu X(\widetilde{x}).\mathtt{L}\big)\langle\widetilde{\ell}\rangle$ where the formal parameters $\widetilde{x}$ stand for the locations used for coordination and $\widetilde{\ell}$ are the initial values. Note that $\mu_\rho\ X.\mathtt{K}$ does not make explicit the set of initial locations but they are so in local types. For this reason, we define projection for a decorated version of global types, where each recursive sub-term $\mu_\rho\ X.\mathtt{K}$ is annotated by a function $\phi : \mathcal{R} \mapsto \mathcal{L}oc$ defined such that $dom(\phi) = \mathsf{roles}(\mathtt{K}) \setminus \{\rho\}$ and for all $\rho \in dom(\phi)$, $\phi(\rho)$ is globally fresh. Such annotations can be automatically added by pre-processing global types so to associate a fresh set of locations to each recursive process. Then, the projection of $\mu^{\phi}_{\rho'}\ X.\mathtt{K}'$ onto ρ depends on whether ρ coordinates the recursion (i.e., $\rho = \rho'$) or not. When ρ is not the coordinator, the recursive process needs just one location x to await for either `stop` or a new location y for the next iteration. Note that the body of the recursion $\mathtt{K}'$ is then projecting by considering an extended version of η where process variable X is parameterised with the received location y. The initial value of x is fixed according to ϕ (i.e., $\phi\rho$). Differently, when ρ coordinates the recursion, the projection generates a process variable that has several parameters, i.e., one location x_i for each passive role. In this case the body of the recursion consists of two branches: one that communicates the termination of the recursion to each participant, and the other one executes the body of the recursion after distributing fresh localities to each participants. Recursion parameters are initialised analogously. Finally, a process variable X is projected as its parameterised version $X\langle\eta X\rangle$, where the value of parameters are established according to η.

4 Semantic Correspondence

This section establishes the correspondence between the denotational semantics of global types and the operational semantics of local types. The partial order on the events of a pomset yields an interpretation of linear executions in terms of *linearisations* similar to interleaved semantics of concurrent systems. Intuitively a linearisation of a pomset r is a sequence of the events $\mathcal{E}_r$ that preserves the pomset's order $\leq_r$. We show that traces of projections of a global type correspond to linearisations of its pomset semantics and that for each linearisation in the pomset semantics there is a system executing a corresponding trace. We first formalise the notion of linearisation.

Given a pomset r and a set of its events $E \subseteq \mathcal{E}_r$, a permutation $e_1 \cdots e_n$ of the events in E is a *linearisation of* r if

– $E \subseteq \mathcal{E}_r$ preserves $\leq_r$ namely $\forall 1 \leq i < j \leq n : \neg(e_j \leq_r e_i)$

- each event in $\mathcal{E}_r$ corresponding to an access of an interaction is in E, namely if $e \in \mathcal{E}_r$ and the tuple type in $\lambda_r(e)$ is of the form $\rho \cdot \mathtt{t}$ then $e \in E$
- each output event in $\mathcal{E}_r$ is in E and, letting $I(e)$ be the set of events in $\mathcal{E}_r$ which are labelled by inputs of a tuple type matching the one in $\lambda_r(e)$, $I(e) \cap E = \emptyset \iff I(e) = \emptyset$
- accesses in $e_1 \cdots e_n$ are preceded by a matching output, namely (i) for each $1 \le i \le n$ if e_i accesses $\mathtt{t}$ at $\mathtt{l}$ then there is some j with $1 \le j < i$ such that e_j outputs $\mathtt{t}'$ at $\mathtt{l}$ with $\mathtt{t}' \bowtie \mathtt{t}$ and (ii) for all h such that $j < h < i$ if e_h inputs $\mathtt{t}''$ at $\mathtt{l}$ then $\neg(\mathtt{t}' \bowtie \mathtt{t}'')$.

Fix a sequence

$$^{[\text{-}]}\pi_1 \ldots {}^{[\text{-}]}\pi_n \tag{3}$$

of labels of events (decorations are immaterial hence omitted in the following). We say that (3) is in *normal form* if the defined names of any two generating labels are disjoint; formally, for all $1 \le i \ne j \le n$

$$\pi_i \text{ generates } \mathtt{t}_i \text{ at } \mathtt{l} \wedge \pi_j \text{ generates } \mathtt{t}_j \text{ at } \mathtt{l} \implies dn(\mathtt{t}_i) \cap dn(\mathtt{t}_j) = \emptyset$$

Also, for $1 \le i < j \le n$, we say that π_j *is in the scope of* π_i if π_i generates $\mathtt{t}_i$ at $\mathtt{l}$ and π_j generates $\mathtt{t}_j$ at $\mathtt{l}$ with $\mathtt{t}_i \bowtie \mathtt{t}_j$ and $\forall i < h < j\ :\ \pi_h$ generates $\mathtt{t}_h$ at $\mathtt{l} \implies \neg(\mathtt{t}_h \bowtie \mathtt{t}_j)$. Without loss of generality we can assume that each sequence like (3) is in normal form (since we can rename all defined names generated by some π_i and the names of the labels π_j in their scope).

Let $\pi \vdash \alpha$ hold iff

$$\begin{cases} (\pi = \rho\,!\,(\mathtt{t})\,@\,\mathtt{l} \vee \pi = \rho\,!\,\mathtt{t}\,@\,\mathtt{l}) & \wedge\ \alpha = \rho : \mathtt{t}'\,!\,\mathtt{l} \\ \pi = \rho\,?\,(\mathtt{t})\,@\,\mathtt{l} & \wedge\ \alpha = \rho : (\mathtt{t}')\,?\,\mathtt{l} \\ \pi = \rho\,?\,\mathtt{t}\,@\,\mathtt{l} & \wedge\ \alpha = \rho : \mathtt{t}'\,?\,\mathtt{l} \end{cases}$$
$$\text{and } \exists \sigma : dn(\mathtt{t}) \to fn(\mathtt{t}')\ :\ \downarrow\!\mathtt{t}\sigma = \mathtt{t}'$$

This definition extends to sequences (3) with $n \ge 1$ as follows: $^{[\text{-}]}\pi_1 \cdots {}^{[\text{-}]}\pi_n \vdash \alpha_1 \cdots \alpha_n$ if $n = 1$ and $\pi_1 \vdash \alpha_1$ or $n > 1$ and

$$\pi_1 \vdash \alpha_1 \wedge \forall \sigma : dn(\mathtt{t}) \to fn(\mathtt{t}')\ :\ \downarrow\!\mathtt{t}\sigma = \mathtt{t}' \implies ({}^{[\text{-}]}\pi_2 \cdots {}^{[\text{-}]}\pi_n)\sigma \vdash \alpha_2 \cdots \alpha_n$$

where $\mathtt{t}$ is the tuple in π and $\mathtt{t}'$ is the one in α_1.

The $\mathtt{K}$*-specification* of a given a global type $\mathtt{K}$ is a specification Δ made of the projections of $\mathtt{K}$ only; formally

(ii) $\rho : \mathtt{L} \in \Delta$ iff $\rho \in \mathsf{roles}(\mathtt{K})$ and $\mathtt{L} = \mathtt{K}\downharpoonright_\rho$, and
(i) Δ has no tuple.

Our main results give a correspondence between the pomset semantics of a global type $\mathtt{K}$ and its $\mathtt{K}$-specification.

Theorem 1. *Given a well-formed global type $\mathtt{K}$, for all $r \in [\![\mathtt{K}]\!]$ there is $\mathtt{K}$-specification Δ such that for all linearisations $e_1 \cdots e_n$ of r there is $\Delta \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}$ such that $\lambda_r(e_1) \cdots \lambda_r(e_n) \vdash \alpha_1 \cdots \alpha_n$.*

Proof (Sketch). The proof shows that the specification $\Delta = \left(\rho : \mathtt{K}\downarrow_\rho\right)_{\rho \in \mathsf{roles}(\mathtt{K})}$ satisfies the property in the conclusion of the statement above. By induction on the structure of $\mathtt{K}$, one shows that

– each output event is matched by an application on Δ of the [LOut] rule in Fig. 2, which adds a tuple type to the specification
– each input or read event has a correspondent transition in Δ from the receiving role according to rules [LIn] and [LRd] respectively; note that (cf. Fig. 2) in the former case the tuple type is removed from the specification.

For input and read events, the existence of the substitution required by the $\vdash$ relation is guaranteed by the hypothesis of rules [LIn] and [LRd]. The above follows immediately in the cases of prefixes. And, in the case of sum the thesis follows by induction because the semantics of a choice is the union of the semantics of each branch. □

Theorem 2. *Let Δ be a $\mathtt{K}$-specification with $\mathtt{K}$ a well-formed global type. For all $\Delta \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}$ there is a linearisation $e_1 \cdots e_n$ of a pomset $r \in [\![\mathtt{K}]\!]$ such that $\lambda_r(e_1) \cdots \lambda_r(e_n) \vdash \alpha_1 \cdots \alpha_n$.*

Proof (Sketch). As for Theorem 1, the proof goes by induction on the structure of $\mathtt{K}$. Guided by the structure of $\mathtt{K}$, we can relate the application of the rules of Fig. 2 with the pomset semantics of the projections. □

5 Conclusions

This paper, a modest attempt to thank Rocco for his work and friendship, addresses the following question:

> What notion of behavioural types corresponds to Linda-based coordination mechanisms?

To answer such question we advocate Klaim-based global and local types, dubbed klaimographies. Klaim has been designed to program distributed systems consisting of processes interacting via multiple distributed tuple spaces.

For simplicity, we have neglected code mobility, a distinctive feature of Klaim. Accommodating the mobility mechanism of Klaim would require to control the multiplicity of running instances and to generalise the well-formedness conditions to dynamically spawned processes. A further challenge, would be to include mobility of processes-as-values featured by Klaim, which shares many similarities with session delegation. However, this can be associated to control-driven problems. These challenges are scope for future work.

We have also not considered parallel types. A simple way to compose klaimographies in parallel would be to follow standard approaches restricting roles on single threads and disjoint tuple spaces. We consider this not very interesting, and plan to explore more expressive settings for parallel types such as the one in [11,12]. In particular, we conjecture that to add parallel composition $\mathtt{K} \mid \mathtt{K}'$ of klaimographies it is enough to require that $\neg(\mathtt{t} \bowtie \mathtt{t}')$ for all $(\mathtt{t}, \mathtt{l}) \in \mathtt{K}$, $(\mathtt{t}', \mathtt{l}) \in \mathtt{K}'$. This condition is the counterpart of the *well-forkedness* condition of [11,12], that requires that different threads of a choregraphy have disjoing input actions.

Klaim has been extended with several features designed on theoretical foundations and implemented in a suite of prototypes [1]. On the one hand, klaimographies share similarities with standard behavioural types centred on point-to-point channel based communications, on the other hand they also have some peculiarities, some of which we highlighted here.

The closest work to our is [5], which develops the initial proposal on parameterised choreographies in [16,7]. Notably, [5] is the first work to support indexed roles and to statically infer the participants inhabiting them. The main difference with the approach in [5] is that klaimographies do not focus on processes, but rather on data. We envisage behavioural types as specifications of how to guarantee general properties of tuple spaces. For instance, take the marketplace example (cf. Example 5), one would like to check properties such as

> for each tuple type $\mathtt{t} = i : \mathtt{str} \cdot p : \mathtt{int} \cdot \nu l : \mathtt{loc}$ consumed from locality $\mathtt{m}$ either a tuple type $\mathtt{sold}$ is eventually generated at locality l or $\mathtt{t}$ is eventually generated at $\mathtt{m}$.

Such property does not concern typical properties controlled by behavioural types (e.g., progress of processes, message orphanage, or unspecified reception).

As scope for future work, we aim to characterise the (classes of) properties of interest that klaimographies enforce. We conjecture that the well-formedness conditions defined here are strong enough to guarantee the property above. Another interesting line of research is to identify typing principles for Klaim processes. We believe that klaimographies can enable the possibility that a same process enacts different roles. For instance, considering again the marketplace example, a process can act both as seller and as buyer.

We have adopted a few simplifying assumptions. Other variants seem rather interesting. For instance, guards of sums could be autonomous inputs and not just consuming interactions, or even read-only access prefixes. Relaxing the constraint that read-only tuples cannot generate, would lead to a sort of multi-cast mechanism of fresh localities. We plan to study those variants in future work.

References

1. L. Bettini, V. Bono, R. De Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The Klaim project: Theory and practice. In C. Priami, editor, *Global Computing*, volume 2874 of *LNCS*, pages 88–150. Springer, 2003.
2. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In P. Gastin and F. Laroussinie, editors, *CONCUR 2010*, volume 6269 of *LNCS*, pages 162–176. Springer, 2010.
3. L. Bocchi, H. C. Melgratti, and E. Tuosto. Resolving non-determinism in choreographies. In Z. Shao, editor, *ESOP 2014*, volume 8410 of *LNCS*, pages 493–512. Springer, 2014.

4. G. Castagna, M. Dezani-Ciancaglini, and L. Padovani. On global types and multiparty session. *Logical Methods in Computer Science*, 8(1), 2012.
5. D. Castro, R. Hu, S. Jongmans, N. Ng, and N. Yoshida. Distributed programming using role-parametric session types in go: Statically-typed endpoint APIs for dynamically-instantiated communication structures. In *POPL 2019*, volume 3 of *PACMPL*, pages 29:1–29:30. ACM, 2019.
6. R. De Nicola, G. L. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
7. P.-M. Denielou, N. Yoshida, A. Bejleri, and R. Hu. Parameterised multiparty session types. *Logical Methods in Computer Science*, 8(4), 2012.
8. M. Dezani-Ciancaglini and U. de' Liguoro. Sessions and session types: An overview. In C. Laneve and J. Su, editors, *WS-FM 2009*, volume 6194 of *LNCS*, pages 1–28. Springer, 2010.
9. H. Gaifman and V. R. Pratt. Partial order models of concurrency and the computation of functions. In *LICS 1987*, pages 72–85. IEEE Computer Society, 1987.
10. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
11. R. Guanciale and E. Tuosto. An abstract semantics of the global view of choreographies. In M. Bartoletti, L. Henrio, S. Knight, and H. T. Vieira, editors, *ICE 2016*, volume 223 of *EPTCS*, pages 67–82, 2016.
12. R. Guanciale and E. Tuosto. Semantics of global views of choreographies. *Logic and Algebraic Methods in Programming*, 95:17–40, 2018.
13. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *ESOP 1998*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.
14. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In G. C. Necula and P. Wadler, editors, *POPL 2008*, pages 273–284, 2008.
15. H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro. Foundations of session types and behavioural contracts. *ACM Computing Survey*, 49(1):3:1–3:36, 2016.
16. N. Yoshida, P. Deniélou, A. Bejleri, and R. Hu. Parameterised multiparty session types. In C. L. Ong, editor, *FoSSaCS 2010*, volume 6014 of *LNCS*, pages 128–145. Springer, 2010.