Procedural Choreographic Programming

Luís Cruz-Filipe, Fabrizio Montesi

University of Southern Denmark, Department of Mathematics and Computer Science, Campusvej 55, 5230 Odense M, Denmark

Abstract

Choreographic Programming is an emerging paradigm for correct-by-construction concurrent programming based on message passing. Models based on choreographic programming have been successfully developed for different settings where concurrent programming is challenging, including service-oriented computing and cyber-physical systems. However, the general applicability of the paradigm is limited by the current lack of support for reusable procedures, which hinders modularity.

We propose Procedural Choreographies (PC), a choreographic language model with full procedural abstraction. PC includes unbounded process creation and name mobility, yielding a powerful framework for writing correct concurrent algorithms that can be compiled into a process calculus. This expressivity requires a typing discipline to ensure that processes are properly connected when enacting procedures. Connections may form networks of arbitrary graph structures. We develop a formal synthesis procedure that, given a program in PC, generates a correct-by-construction concurrent implementation in terms of a process calculus. We illustrate the expressivity of PC with a series of examples, including parallel streams and parallel computation based on pipelining.

Keywords: Choreographic Programming, Deadlock-freedom, Process Calculi

1. Introduction

Developing correct concurrent software is challenging, because it is hard to reason about multiple simultaneous executions. In the setting of message passing, programming errors can lead to unexpected communications, or systems reaching a deadlock because of mismatched I/O actions. Choreographic Programming (Montesi, 2013) is a paradigm for programming concurrent software based on message passing that is deadlock-free by construction, by using an "Alice and Bob" notation that syntactically prevents mismatched I/O communications in programs (called choreographies), and using an EndPoint Projection

Email addresses: lcf@imada.sdu.dk (Luís Cruz-Filipe), fmontesi@imada.sdu.dk (Fabrizio Montesi)

(EPP for short) to synthesise correct process implementations (Qiu et al., 2007; Carbone et al., 2012; Carbone and Montesi, 2013). The key idea of choreographic notation is to define the desired interactions explicitly, as atomic primitives, rather than doing it indirectly by composing separate I/O actions. This makes communication flows explicit, in a way that recalls the security protocol notation of Needham and Schroeder (1978) and message sequence charts (International Telecommunication Union, 1996). For these reasons, choreographies have been used in: standards, like the Business Process Model and Notation by the Object Management Group (2017) and the Web Services Choreography Description Language (WS-CDL) by the W3C WS-CDL Working Group (2004); formal models for concurrent systems (Qiu et al., 2007; Carbone et al., 2012); experimental languages (Honda et al., 2011; Chor, 2017); and software development methodologies, like Testable Architecture (JBoss Community and Red Hat, 2017).

Since their inception, the link between choreographies and the theory of process calculi was evident (Busi et al., 2006; Bravetti and Zavattaro, 2007; Qiu et al., 2007; Lanese et al., 2008; Carbone et al., 2012). This sparked a fruitful line of research that, much alike to the one on process calculi, can be seen as a "workshop" of choreographic models designed to explore different features. Recently, in particular, research on choreographic programming has gained in breadth, and our knowledge of choreographies is extending rapidly. For example, choreographic programming has been studied in the settings of: service-oriented programming, both for the synthesis of skeleton communication code to be used as a "black box" from independent services (Carbone and Montesi, 2013; Dalla Preda et al., 2014) or for the formalisation of modular systems using behavioural types (Montesi and Yoshida, 2013); runtime adaptation (Dalla Preda et al., 2017); information flow (Lluch-Lafuente et al., 2015); linear logic, where choreographies syntactically describe the process of cut elimination (Carbone et al., 2018); cyber-physical systems (López et al., 2016; López and Heussen, 2017); and decompilation, which intuitively is the right adjoint of EPP (Lange et al., 2015; Cruz-Filipe et al., 2017).

In this work, we focus on another important aspect: procedural programming. Writing procedures that can be arbitrarily instantiated and composed into larger programs is still unsupported in choreographic programming. The absence of full procedural abstraction prevents the creation of libraries that can be reused in different programs.

Example 1. *We discuss a parallel version of merge sort, written as a choreography. We assume that concurrent processes communicate by exchanging messages and possess local memory. In this example, each process stores a list and can use the following local functions:* `split1` *and* `split2`*, respectively returning the first or the second half of the stored list;* `is_small`*, which tests if the stored list has at most one element; and* `merge`*, which combines two sorted lists into one. The following (choreographic) procedure,* `MS`*, implements merge sort on the list*

stored at its parameter process `p`.[1]

```
MS(p) = if p.is_small then 0
        else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
             MS<q1>; MS<q2>; q1.* -> p; q2.* -> p.merge
```

Procedure `MS` *starts by checking whether the list at process* `p` *is small, in which case it does not need to be sorted (*`0` *denotes termination); otherwise,* `p` *starts two other processes* q_1 *and* q_2 *(*`p start q1,q2`*), to which it respectively sends the first and the second half of the list (*`p.split1 -> q1`* and* `p.split2 -> q2`*). We point out that* q_1 *and* q_2 *are process variables, bound by the* `start` *action. The procedure is recursively reapplied to* q_1 *and* q_2*, which independently (concurrently) proceed to ordering their respective sub-lists. When this is done,* `MS` *stores the first ordered half from* q_1 *to* `p` *(*`q1.* -> p`*, where* `*` *retrieves the data stored in* q_1*) and merges it with the ordered sub-list from* q_2 *(*`q2.* -> p.merge`*).*

Procedure `MS` in Example 1 is a simple toy example. Even so, it cannot be written in current choreographic programming models because it uses two unsupported features: *general recursion*, allowing procedure calls to be followed by arbitrary code; and *parametric procedures*, which can be reused with different processes (as in `MS<q1>` and `MS<q2>`). These features are key to many useful algorithmic and communication patterns in practice (like divide-et-impera and map-reduce). Similarly, in the remainder, we explore more realistic and involved examples that illustrate the need for more advanced features, like: mobility of process names, which enables networks with connections that evolve at runtime; the propagation of branching choices among processes; and the support for procedures that take a variable number of process parameters, where the exact number is decided at runtime. As we show through our examples, these features are important in the context of procedural choreographic programming because they allow us to: connect the processes that need to communicate in order to execute a procedure; make processes agree on what procedure they should execute together; and write procedures that define the behaviour of groups of processes of variable size. Mobility and branching have been explored in restricted form in previous work, which is not enough for our setting where the number of processes simultaneously executing is unbounded (for related work, see § 7).

Developing a language model for choreographic programming that supports these features is challenging, because we need to provide an EndPoint Projection (EPP) – EPP is what makes choreographic programming useful, since it is how we synthesise process implementations (which abstractly represent executable code) from choreographies. This challenge is multifaceted. First, EPP should generate code that follows the original choreography (correctness-by-construction) and is deadlock-free, which are two typical properties of choreography-based models. For this, we need to deal with different issues caused by potential

[1] In this work, we use a `monospaced` font for readability of our concrete examples, and other fonts for distinguishing syntactic categories in our formal arguments, as usual.

bad usage of our features of interest, for example: attempted communication between two unconnected processes (e.g., bad usage of name mobility), or disagreement on which processes should enact a procedure (e.g., bad usage of procedure parametricity). Second, EPP should respect the programmer's intention on communications, i.e., the synthesised processes should execute exactly the communications defined in the originating choreography. Concretely, this means that EPP cannot inject extra communications (or remove some) to ensure that all processes are coordinated as needed. If we did that, we would risk adding unexpected overhead – maybe the efficiency of the choreography given as input is relevant – or even unexpected information leaks – maybe it is actually important for the programmer that some processes do not know of some decision. Third, EPP should yield parallel implementations when the algorithm described in a choreography is conceptually parallel. This requires understanding when procedure executions can be parallelised. For instance, in Example 1, the two calls in `MS<q1>; MS<q2>` can be run in parallel because they involve separate processes and are thus non-interfering.

Contributions. We present Procedural Choreographies (PC), a new model for choreographic programming that includes all the ingredients mentioned earlier. PC has a simple syntax, but as we show throughout the remainder, it is expressive enough to write sophisticated concurrent programs (e.g., implementing concurrent data streams or pipelining). The semantics of PC infers automatically safe concurrent executions of choreographic procedures (and choreographic code in general, down to the level of single statements).

PC comes with a typing discipline that prevents wrong procedure composition (e.g., due to dangling process references or wrong parameter usage), by typing the connections that a code block in a choreography requires and produces. Typing ensures that choreographies progress. This discipline supports both decidable type checking and type inference.

We define an EndPoint Projection (EPP) that synthesises concurrent implementations of choreographies in terms of a process calculus, and prove that such implementations are correct (operationally equivalent to the choreography) and deadlock-free. This process calculus is an abstraction of systems where processes refer to one another's locations or identifiers (as in the MPI Forum (2015) or the Internet Protocol).

We illustrate the expressivity of our development through a series of representative examples: parallel sorting (mergesort, quicksort), distributed authentication, parallel streams (a downloader), Gaussian Elimination (which uses pipelining), and Fast Fourier Transform.

All our results hold both for systems that make use of synchronous (rendezvous) communications and for systems based on asynchronous communications. As a consequence, a choreography developed with synchronous communications in mind can be safely reused in a system with asynchronous communications, with no intervention required by the programmer: we simply get a more concurrent implementation, thanks to asynchrony. The extension to the asynchronous setting is a simple adaptation of the technique of Cruz-Filipe and

$$C ::= \eta; C \mid I; C \mid \mathbf{0} \qquad \eta ::= \mathsf{p}.e \texttt{ -> } \mathsf{q}.f \mid \mathsf{p} \texttt{ -> } \mathsf{q}[\ell] \mid \mathsf{p} \,\mathsf{start}\, \mathsf{q}^T \mid \mathsf{p}\colon \mathsf{q} \texttt{ <-> } \mathsf{r}$$
$$\mathscr{D} ::= X(\widetilde{\mathsf{q}^T}) = C, \mathscr{D} \mid \emptyset \qquad I ::= \mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid X\langle \tilde{\mathsf{p}} \rangle \mid \mathbf{0}$$

Figure 1: Procedural Choreographies, Syntax.

Montesi (2017a) to PC, and is therefore only sketched in § 7. The details of the construction can be found in the technical report by Cruz-Filipe and Montesi (2016b).

Publication history. This article combines results previously included in two conference publications. The language of PC, together with its synchronous semantics, type system and EndPoint Projection, was originally published by Cruz-Filipe and Montesi (2017b), while some language extensions and the more complex examples (QuickSort, Gaussian elimination, Fast Fourier transform) were described by Cruz-Filipe and Montesi (2016). The proofs of all results are presented here for the first time: they were only included in the technical report by Cruz-Filipe and Montesi (2016b).

Structure. We describe the syntax and semantics of PC in § 2, and its type system in § 3. § 4 introduces the model that we use to represent process implementations and the corresponding EPP from PC. We present a small extension of PC in § 5 that does not intrinsically change its theory, but allows us to write the more sophisticated examples in § 6. We discuss related work in § 7 before concluding in § 8.

2. Procedural Choreographies (PC)

We begin by introducing the language model of Procedural Choreographies (PC).

2.1. Syntax

A procedural choreography is a pair $\langle \mathscr{D}, C \rangle$, where C is a choreography and $\mathscr{D}$ is a set of procedure definitions, following the syntax displayed in Figure 1. Process names ($\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots$) identify processes that execute concurrently. Each process is equipped with a memory cell that stores a single value of a fixed type. Specifically, we consider a fixed set $\mathbb{T}$ of datatypes (numbers, lists, etc.); each process p stores only values of type $T_{\mathsf{p}} \in \mathbb{T}$.

Statements in a choreography can either be communication actions (η) or compound instructions (I), both of which can have continuations. Term $\mathbf{0}$ is the terminated choreography, which we often omit in examples. We call all terms but $\mathbf{0}; C$ *program terms*, or simply programs, since these form the syntax intended for developers to use for writing programs. Term $\mathbf{0}; C$ is necessary only for the technical definition of the semantics, to capture termination of

procedure calls with continuations, and can appear only at runtime. It is thus called a *runtime term*.

Processes communicate through direct references (names) to each other.[2] In a value communication $\mathsf{p}.e \texttt{ -> } \mathsf{q}.f$, process p sends the result of evaluating expression e to q. The expression e may contain the placeholder $*$, which is replaced at runtime with the data in p's memory. When q receives the value from p, it applies to it the (total) function f and stores the result. The definition of f may also access the contents of q's memory (through $*$). The precise syntax of expressions and functions is left unspecified, as it is essentially immaterial for our development.

In a selection term $\mathsf{p} \texttt{ -> } \mathsf{q}[\ell]$, p communicates to q its choice of label ℓ, which is a constant. This term is intended to propagate information on which internal choice has been made by a process to another (see Remark 3 below).

In term $\mathsf{p}\,\mathsf{start}\,\mathsf{q}^T$, process p spawns the new process q, which stores data of type T. Process name q is bound in the continuation C of $\mathsf{p}\,\mathsf{start}\,\mathsf{q}^T; C$.

Process spawning introduces the need for name mobility. In real-world systems, after execution of $\mathsf{p}\,\mathsf{start}\,\mathsf{q}^T$, p is the only process that knows q's name. Any other process wanting to communicate with q must therefore be first informed of its existence. This is achieved with the introduction term $\mathsf{p}\colon \mathsf{q} \texttt{ <-> } \mathsf{r}$, read "$\mathsf{p}$ introduces q and r" (with p, q and r distinct). As its double-arrow syntax suggests, this action represents *two* communications – one where p sends q's name to r, and another where p sends r's name to q. This is made explicit in § 4.

In a conditional term $\mathsf{if}\,\mathsf{p}.e\,\mathsf{then}\,C_1\,\mathsf{else}\,C_2$, process p evaluates e to choose between the possible continuations C_1 and C_2.

The set $\mathscr{D}$ contains definitions of global procedures. Term $X(\widetilde{\mathsf{q}^T}) = C_X$ defines a procedure X with body C_X, which can be used anywhere in $\langle \mathscr{D}, C \rangle$ – in particular, inside C_X. The names $\tilde{\mathsf{q}}$ are bound to C_X, and they are exactly the free process names in C_X.[3] Each procedure can be defined at most once in $\mathscr{D}$. Term $X\langle \tilde{\mathsf{p}} \rangle$ calls (invokes) procedure X by passing the processes in $\tilde{\mathsf{p}}$ as parameters. All processes in $\tilde{\mathsf{p}}$ must be distinct. Furthermore, procedure calls inside definitions must be guarded, i.e., they can only occur after some other action,

Remark 1. *In PC, we make heavy use of the Barendregt variable convention: all bound variables have distinct names, and those names are distinct from the names of all free variables. We use this convention in two implicit assumptions: (1) we work up to α-equivalence in choreographies, renaming bound variables as needed; (2) in reductions that duplicate bound variables (typically, when expanding procedure calls), we assume that the bound variables are renamed to conform with the convention. For example, if we include the definition*

[2] PC thus easily applies to settings based on actors, objects, or ranks (e.g., MPI).

[3] We could relax this requirement by having $\tilde{\mathsf{q}}$ be a superset of the free process names in C_X. Having equality simplifies our technical development later.

$X(\mathsf{p}) = \mathsf{p}\,\mathsf{start}\,\mathsf{r}; X(\mathsf{r})$, *then unfolding* X *twice in the choreography* $X(\mathsf{q})$ *yields, e.g.,* $\mathsf{q}\,\mathsf{start}\,\mathsf{r}; \mathsf{r}\,\mathsf{start}\,\mathsf{r}'; X(\mathsf{r}')$, *rather than* $\mathsf{q}\,\mathsf{start}\,\mathsf{r}; \mathsf{r}\,\mathsf{start}\,\mathsf{r}; X(\mathsf{r})$.

Example 2. *Recall procedure* `MS` *from our merge sort example in the Introduction (Example 1). If we annotate the parameter* p *and the started processes* q_1 *and* q_2 *with a type, e.g.,* $\mathbf{List}(T)$ *for some* T *(the type of lists containing elements of type* T*), then* `MS` *is a valid procedure definition in PC, as long as we allow two straightforward syntactic conventions: (i)* $\mathsf{p}\,\mathsf{start}\,\widetilde{\mathsf{q}^T}$ *stands for the sequence* $\mathsf{p}\,\mathsf{start}\,\mathsf{q}_1^{T_1}; \ldots; \mathsf{p}\,\mathsf{start}\,\mathsf{q}_n^{T_n}$*; (ii) a communication of the form* $\mathsf{p}.e \to \mathsf{q}$ *stands for* $\mathsf{p}.e \to \mathsf{q}.\mathsf{id}$*, where* id *is the identity function: this action simply sets the content of* q *to the value received from* p*. We adopt these conventions also in the remainder.*

Remark 2 (Design choices)**.** *We comment on two of our design choices.*

The introduction action ($\mathsf{p} : \mathsf{q} \leftrightarrow \mathsf{r}$*) requires a three-way synchronization, essentially performing two communications. The alternative development of PC with asymmetric introduction (an action* $\mathsf{p} : \mathsf{q} \to \mathsf{r}$ *whereby* p *sends* q*'s name to* r*, but not conversely) would be very similar. Since in our examples we always perform introductions in pairs, the current choice makes the presentation easier.*

The restriction that each process stores only one value of a fixed type is, in practice, a minor constraint. As in Example 2, types can be tuples or lists, which mimics storing several values. Also, a process can create new processes with different types – so we can encode changing the type of p *by having* p *create a new process* p' *and then continuing the choreography with* p' *instead of* p*.*

Remark 3 (Label Selection)**.** *We motivate the need for selections (*$\mathsf{p} \to \mathsf{q}[\ell]$*). Consider the choreography* $\mathsf{if}\,\mathsf{p}.\mathsf{coinflip}\,\mathsf{then}\,(\mathsf{p}.* \to \mathsf{r})\,\mathsf{else}\,(\mathsf{r}.* \to \mathsf{p})$*. Here,* p *flips a coin to decide whether to send a value to* r *or to receive a value from* r*. Since processes run independently and share no data, only* p *knows which branch of the conditional will be executed; but this information is essential for* r *to decide on its behaviour. To propagate* p*'s decision to* r*, we use selections:*

$$\mathsf{if}\,\mathsf{p}.\mathsf{coinflip}\,\mathsf{then}\,(\mathsf{p} \to \mathsf{r}[\mathrm{L}]; \mathsf{p}.* \to \mathsf{r})\,\mathsf{else}\,(\mathsf{p} \to \mathsf{r}[\mathrm{R}]; \mathsf{r}.* \to \mathsf{p})$$

Now r *receives a label reflecting* p*'s choice, and can use it to decide what to do.*

This information propagation is essential for compilation (see § 4): the first choreography above is not projectable, whereas the second one is. Selections can be inferred automatically, and thus could be removed from the user syntax, but it is useful to be able to specify them manually (see Remark 8). This aspect is found repeatedly in the choreography models by Carbone et al. (2012), Carbone and Montesi (2013), Coppo et al. (2016) and Cruz-Filipe and Montesi (2016a). See also Example 5 at the end of this section.

2.2. Semantics

We define a reduction semantics $\to_{\mathscr{D}}$ for PC, parameterised over the set $\mathscr{D}$ (Figure 2). Given a choreography C, we model the state of its processes with a state function σ, where $\sigma(\mathsf{p})$ denotes the value stored in p. The domain of σ

$$\frac{\mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{q} \quad e \downarrow^{\mathsf{p}}_{\sigma} v \quad f(v) \downarrow^{\mathsf{q}}_{\sigma} w}{G, \mathsf{p}.e \texttt{ -> } \mathsf{q}.f; C, \sigma \to_{\mathscr{D}} G, C, \sigma[\mathsf{q} \mapsto w]} \lfloor \text{C|Com} \rceil \qquad \frac{\mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{q}}{G, \mathsf{p} \texttt{ -> } \mathsf{q}[\ell]; C, \sigma \to_{\mathscr{D}} G, C, \sigma} \lfloor \text{C|Sel} \rceil$$

$$\frac{}{G, \mathsf{p} \,\mathsf{start}\, \mathsf{q}^{T}; C, \sigma \to_{\mathscr{D}} G \cup \{\mathsf{p} \leftrightarrow \mathsf{q}\}, C, \sigma[\mathsf{q} \mapsto \bot_T]} \lfloor \text{C|Start} \rceil$$

$$\frac{\mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{q} \quad \mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{r}}{G, \mathsf{p}\texttt{: } \mathsf{q} \texttt{ <-> } \mathsf{r}; C, \sigma \to_{\mathscr{D}} G \cup \{\mathsf{q} \leftrightarrow \mathsf{r}\}, C, \sigma} \lfloor \text{C|Tell} \rceil$$

$$\frac{i = 1 \text{ if } e \downarrow^{\mathsf{p}}_{\sigma} \mathsf{true}, \quad i = 2 \text{ otherwise}}{G, (\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2); C, \sigma \to_{\mathscr{D}} G, C_i \fatsemi C, \sigma} \lfloor \text{C|Cond} \rceil$$

$$\frac{C_1 \preceq_{\mathscr{D}} C_2 \quad G, C_2, \sigma \to_{\mathscr{D}} G', C_2', \sigma' \quad C_2' \preceq_{\mathscr{D}} C_1'}{G, C_1, \sigma \to_{\mathscr{D}} G', C_1', \sigma'} \lfloor \text{C|Struct} \rceil$$

Figure 2: Procedural Choreographies, Semantics.

is the set $\mathsf{pn}(C)$ containing all free process names in C. (As C is executed, the domain of σ is extended by the processes that are created.) We assume that each type $T \in \mathbb{T}$ has a special value $\bot_T$, representing an uninitialised process state. We also use a connection graph G, keeping track of which processes know each other. In the rules, $\mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{q}$ denotes that G contains an edge between p and q, and $G \cup \{\mathsf{p} \leftrightarrow \mathsf{q}\}$ denotes the graph obtained from G by adding an edge between p and q (if missing).

Executing a communication action $\mathsf{p}.e$ -> $\mathsf{q}.f$ in rule $\lfloor$C|Com$\rceil$ requires that: p and q are connected in G; e is well typed; and the type of e matches that expected by the function f at the receiver. The last two conditions are encapsulated in the conditions $e \downarrow^{\mathsf{p}}_{\sigma} v$ and $f(v) \downarrow^{\mathsf{q}}_{\sigma} w$. The term $e \downarrow^{\mathsf{p}}_{\sigma} v$, read "$e$ evaluates to v under σ at process p", denotes the result of evaluating the expression obtained by replacing the placeholder $*$ in e with the value $\sigma(p)$. Choreographies can thus deadlock (be unable to reduce) because of errors in the programming of communications; this issue is addressed by our typing discipline in § 3.

Rule $\lfloor$C|Sel$\rceil$ defines selection as a no-op for choreographies (see Remark 3).

Rule $\lfloor$C|Start$\rceil$ models the creation of a process. In the reductum, the starter and started processes are connected and can thus communicate with each other. This rule also extends the domain of the state function σ accordingly. Rule $\lfloor$C|Tell$\rceil$ captures name mobility, creating a connection between two processes q and r when they are introduced by a process p connected to both.

Rule $\lfloor$C|Cond$\rceil$ uses the auxiliary operator $\fatsemi$ to obtain a reductum in the syntax of PC regardless of the forms of the branches C_1 and C_2 and the continuation C. This operator is defined as follows.

$$\mathbf{0} \fatsemi C = C \qquad \eta \fatsemi C = \eta; C \qquad I \fatsemi C = I; C \qquad (C_1; C_2) \fatsemi C = C_1; (C_2 \fatsemi C)$$

An important characteristic of this operator is that it extends the scope of bound names: in $C \fatsemi C'$, any name p bound in C has its scope extended also to C'.

$$\frac{}{\mathbf{0}; C \preceq_{\mathscr{D}} C}\ \lfloor\text{C|End}\rceil \qquad \frac{X(\widetilde{\mathsf{q}^T}) = C_X \in \mathscr{D}}{X\langle\tilde{\mathsf{p}}\rangle; C \preceq_{\mathscr{D}} C_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \fatsemi C}\ \lfloor\text{C|Unfold}\rceil$$

$$\frac{\mathsf{pn}(\eta) \# \mathsf{pn}(\eta')}{\eta; \eta' \equiv_{\mathscr{D}} \eta'; \eta}\ \lfloor\text{C|Eta-Eta}\rceil \qquad \frac{\mathsf{pn}(I) \# \mathsf{pn}(I')}{I; I' \equiv_{\mathscr{D}} I'; I}\ \lfloor\text{C|I-I}\rceil \qquad \frac{\mathsf{pn}(I) \# \mathsf{pn}(\eta)}{\eta; I \equiv_{\mathscr{D}} I; \eta}\ \lfloor\text{C|I-Eta}\rceil$$

$$\frac{\mathsf{p} \notin \mathsf{pn}(\eta)}{\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ (\eta; C_1)\ \mathsf{else}\ (\eta; C_2) \equiv_{\mathscr{D}} \eta; \mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2}\ \lfloor\text{C|Eta-Cond}\rceil$$

$$\frac{\mathsf{p} \notin \mathsf{pn}(I)}{\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ (I; C_1)\ \mathsf{else}\ (I; C_2) \equiv_{\mathscr{D}} I; \mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2}\ \lfloor\text{C|I-Cond}\rceil$$

$$\frac{\mathsf{p} \neq \mathsf{q}}{\begin{array}{c}\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ (\mathsf{if}\ \mathsf{q}.e'\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2)\ \mathsf{else}\ (\mathsf{if}\ \mathsf{q}.e'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2') \\ \equiv_{\mathscr{D}} \\ \mathsf{if}\ \mathsf{q}.e'\ \mathsf{then}\ (\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_1')\ \mathsf{else}\ (\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_2\ \mathsf{else}\ C_2')\end{array}}\ \lfloor\text{C|Cond-Cond}\rceil$$

Figure 3: Procedural Choreographies, Structural Precongruence $\preceq$.

This scope extension is capture-avoiding, as the use of the Barendregt variable convention guarantees that p does not occur in C'.[4]

Rule $\lfloor$C|Struct$\rceil$ uses structural precongruence $\preceq_{\mathscr{D}}$, which is defined by the rules in Figure 3. The first two rules deal with garbage collection of $\mathbf{0}$ (rule $\lfloor$C|End$\rceil$) and unfolding of procedure calls (rule $\lfloor$C|Unfold$\rceil$, again using the $\fatsemi$ operator defined above.) The remaining rules are symmetric, and formalise the notion of parallelism in PC – recalling out-of-order execution, as detailed in Example 3 below. In these rules, we write $C \equiv_{\mathscr{D}} C'$ for $C \preceq_{\mathscr{D}} C'$ and $C' \preceq_{\mathscr{D}} C$, and $A \# B$ for the requirement that the two sets A and B are disjoint. Intuitively, these rules state that actions involving disjoint sets of processes can be swapped (and therefore executed in any order), modelling that processes run independently of one another.

Case in point, rule $\lfloor$C|Eta-Eta$\rceil$ permutes two communications performed by processes that are all distinct. For example, $\mathsf{p}.* \to \mathsf{q}; \mathsf{r}.* \to \mathsf{s} \equiv_{\mathscr{D}} \mathsf{r}.* \to \mathsf{s}; \mathsf{p}.* \to \mathsf{q}$ because these two communications are non-interfering, but $\mathsf{p}.* \to \mathsf{q}; \mathsf{q}.* \to \mathsf{s} \not\equiv_{\mathscr{D}} \mathsf{q}.* \to \mathsf{s}; \mathsf{p}.* \to \mathsf{q}$: since the second communication causally depends on the first (both involve q).

This reasoning is extended to instructions in rule $\lfloor$C|I-I$\rceil$; in particular, procedure calls that share no arguments can be swapped. This is sound, as a procedure can only refer to processes that are either passed as arguments or started inside its body, and the latter cannot be leaked to the original call site. Thus, any actions obtained by unfolding the first procedure call necessarily in-

[4]The reader might wonder why we did not introduce general sequential composition – $C_1; C_2$ – as a primitive. Including this primitive would add no expressivity, but would make the definition of our semantics significantly more complex. In particular, the rules for swapping independent actions, see below, are much easier to define structurally with our syntax.

volve different processes than those obtained by unfolding the second one. As Example 3 below also shows, calls to the same procedure can be exchanged, since X and Y need not be distinct. The remaining rules follow similar intuitions, allowing actions to be moved inside or outside both branches of a conditional, or switching independent nested conditionals.

Example 3. *In our merge sort example, structural precongruence $\preceq_{\mathscr{D}}$ allows the recursive calls* `MS<q1>` *and* `MS<q2>` *to be exchanged. Furthermore, after the calls are unfolded, their code can be interleaved in any way.*

This example exhibits map-reduce behaviour: each new process receives its input, runs independently from all others, and then sends its result to its creator.

Example 4. *Our semantics captures also more fine-grained parallelism. For example, we can sometimes swap communications from procedure calls that share process names. Consider the procedure*

```
auth(c,a,r,l) = c.creds -> a.rCreds;
                a.chk -> r.res; a.log -> l.app
```

Client `c` *sends its credentials to an authentication server* `a`*, which stores the result of authentication in* `r` *and appends a log of this operation at process* `l`*. In the choreography* `auth<c,a1,r1,l>; auth<c,a2,r2,l>`*, a client* `c` *authenticates at two different authentication servers* `a1` *and* `a2`*. After unfolding the two calls, rule* $\lfloor C|Eta\text{-}Eta \rceil$ *can be used to yield the following interleaving:*

```
c.creds -> a1.rCreds; c.creds -> a2.rCreds;
a2.chk -> r2.res; a1.chk -> r1.res;
a1.log -> l.app; a2.log -> l.app
```

Thus, the two authentications can proceed in parallel. Observe that the logging operations cannot be swapped, since they use the same logging process `l`*.*

Example 5. *A more sophisticated example involves modularly composing different procedures that take multiple parameters. Here, we write a choreography where a client* `c` *downloads a collection of files from a server* `s`*. Files are downloaded in parallel via streaming, by having the client and the server each create subprocesses to handle the transfer of each file. Thus, the client can request and start downloading each file without waiting for previous downloads to finish.*

```
par_download(c,s) = if c.more
  then c -> s [more]; c start c'; s start s';
       s: c <-> s'; c.top -> s'; pop<c>;
       c: c' <-> s'; download<c',s'>;
       par_download<c,s>; c'.file -> c.store
  else c -> s [end]
```

At the start of `par_download`*, the client* `c` *checks whether it wants to download more files and informs the server* `s` *of the result via a label selection. In the affirmative case, the client and the server start two subprocesses,* $\mathtt{c}'$ *and* $\mathtt{s}'$ *respectively, and the server introduces* `c` *to* $\mathtt{s}'$ *(*`s: c <-> s'`*). The client* `c` *sends to* $\mathtt{s}'$ *the name of the file to download (*`c.top -> s'`*) and removes it from its collection, using procedure* `pop` *(omitted), afterwards introducing its own subprocess* $\mathtt{c}'$ *to* $\mathtt{s}'$*. The*

file download is handled by c′ *and* s′ *(using procedure* download*), while* c *and* s *continue operating (*par_download<c,s>*). Finally,* c′ *waits until* c *is ready to store the downloaded file.*

Procedure download *has a similar structure. It implements a stream where a file is sequentially transferred in chunks from a process* s *to another process* c*.*

```
download(c,s) = if s.more
  then s -> c [more]; s.next -> c.app; pop<s>; download<c,s>
  else s -> c [end]
```

The implementation of par_download *exploits out-of-order execution considerably. All calls to* download *are made with disjoint sets of parameters (processes), and can thus be fully parallelised: many instances of* download *run at the same time, each one implementing a (sequential) stream. Due to our semantics, we effectively end up executing many streaming behaviours in parallel.*

We can even compose par_download *with* auth*, such that we execute the parallel download only if the client can successfully authenticate with an authentication server* a*. Below, we use the shortcut* $\mathsf{p} \to \tilde{\mathsf{q}}[\ell]$ *for* $\mathsf{p} \to \mathsf{q}_1[\ell]; \ldots; \mathsf{p} \to \mathsf{q}_n[\ell]$*.*

```
auth<c,a,r,l>; if r.ok then r -> c,s[ok]; par_download<c,s>
                       else r -> c,s[ko]
```

3. Typing and Deadlock-Freedom

As we saw in the previous section, choreographies in PC may get stuck because of problems with communication actions. We now give a typing discipline for PC, to check that (a) the types of functions and processes are respected by communications and (b) processes that need to communicate are first properly introduced (or connected). Regarding (b), two processes created independently can communicate only after they receive the names of each other. For instance, in Example 5, the execution of download<c',s'> would get stuck if c' and s' were not properly introduced in par_download, since our semantics requires them to be connected.

Typing judgements have the form $\Gamma; G \vdash C \triangleright G'$, read "$C$ is well-typed according to Γ, and running C with a connection graph that contains G yields a connection graph that includes G'". Typing environments Γ are used to track the types of processes and procedures; they are defined as:

$$\Gamma ::= \emptyset \mid \Gamma,\, \mathsf{p} : T \mid \Gamma,\, X(\widetilde{\mathsf{q}^T}) : G \triangleright G' \,.$$

A typing $\mathsf{p} : T$ states that process p stores values of type T, and a typing $X(\widetilde{\mathsf{q}^T}) : G \triangleright G'$ records the effect of the body of X on graph G, given the names and types of the arguments of X.

The rules for deriving typing judgements are given in Figure 4. We assume standard typing judgements for functions and expressions, and write $* : T \vdash_{\mathbb{T}} e : T$ and $* : T_1 \vdash_{\mathbb{T}} f : T_2 \to T_3$ meaning, respectively "e has type T assuming that $*$ has type T" and "f has type $T_2 \to T_3$ assuming that $*$ has type T_1". Verifying that communications respect the expected types is straightforward,

$$\frac{\mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{q} \quad \Gamma \vdash \mathsf{p} : T_{\mathsf{p}}, \mathsf{q} : T_{\mathsf{q}} \quad \Gamma; G \vdash C \triangleright G' \quad * : T_{\mathsf{p}} \vdash_{\mathbb{T}} e : T_1 \quad * : T_{\mathsf{q}} \vdash_{\mathbb{T}} f : T_1 \to T_{\mathsf{q}}}{\Gamma; G \vdash \mathsf{p}.e \texttt{ -> } \mathsf{q}.f; C \triangleright G'} \lfloor \mathrm{T}|\mathrm{Com} \rceil$$

$$\frac{\mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{q} \quad \Gamma; G \vdash C \triangleright G'}{\Gamma; G \vdash \mathsf{p} \texttt{ -> } \mathsf{q}[\ell]; C \triangleright G'} \lfloor \mathrm{T}|\mathrm{Sel} \rceil \qquad \frac{\mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{q} \quad \mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{r} \quad \Gamma; G \cup \{\mathsf{q} \leftrightarrow \mathsf{r}\} \vdash C \triangleright G'}{\Gamma; G \vdash \mathsf{p}\texttt{: } \mathsf{q} \texttt{ <-> } \mathsf{r}; C \triangleright G'} \lfloor \mathrm{T}|\mathrm{Tell} \rceil$$

$$\frac{\Gamma, \mathsf{q} : T; G \cup \{\mathsf{p} \leftrightarrow \mathsf{q}\} \vdash C \triangleright G'}{\Gamma; G \vdash \mathsf{p} \,\mathsf{start}\, \mathsf{q}^T; C \triangleright G'} \lfloor \mathrm{T}|\mathrm{Start} \rceil \qquad \frac{}{\Gamma; G \vdash \mathbf{0} \triangleright G} \lfloor \mathrm{T}|\mathrm{End} \rceil \qquad \frac{\Gamma; G \vdash C \triangleright G'}{\Gamma; G \vdash \mathbf{0}; C \triangleright G'} \lfloor \mathrm{T}|\mathrm{EndSeq} \rceil$$

$$\frac{\Gamma \vdash \mathsf{p} : T \quad * : T \vdash_{\mathbb{T}} e : \mathsf{bool} \quad \Gamma; G \vdash C_i \triangleright G_i \quad \Gamma; G_1 \cap G_2 \vdash C \triangleright G'}{\Gamma; G \vdash (\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2); C \triangleright G'} \lfloor \mathrm{T}|\mathrm{Cond} \rceil$$

$$\frac{\Gamma \vdash X(\widetilde{\mathsf{q}^T}) : G_X \triangleright G'_X \quad \Gamma \vdash \mathsf{p}_i : T_i \quad G_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \subseteq G \quad \Gamma; G \cup (G'_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}]) \vdash C \triangleright G'}{\Gamma; G \vdash X\langle\tilde{\mathsf{p}}\rangle; C \triangleright G'} \lfloor \mathrm{T}|\mathrm{Call} \rceil$$

Figure 4: Procedural Choreographies, Typing Rules.

using the connection graph G to track which processes have been introduced to each other. In rule $\lfloor \mathrm{T}|\mathrm{Start} \rceil$, we implicitly use the fact that q does not occur in G, which follows from our use of the Barendregt convention. The final graph G' is required to deal with procedure calls (rule $\lfloor \mathrm{T}|\mathrm{Call} \rceil$), and all other rules leave it unchanged.

To type a procedural choreography, we need to type its set of procedure definitions $\mathscr{D}$. We write $\Gamma \vdash \mathscr{D}$ if: for each $X(\widetilde{\mathsf{q}^T}) = C_X \in \mathscr{D}$, there is exactly one typing $X(\widetilde{\mathsf{q}^T}) : G_X \triangleright G'_X \in \Gamma$, and this typing is such that $\Gamma, \widetilde{\mathsf{q} : T}, G_X \vdash C_X \triangleright G''_X$, where G'_X is G''_X restricted to the processes in $\tilde{\mathsf{q}}$.

We say that $\Gamma \vdash \langle \mathscr{D}, C \rangle$ if $\Gamma, \Gamma_{\mathscr{D}}; G_C \vdash C \triangleright G'$ for some $\Gamma_{\mathscr{D}}$ such that $\Gamma_{\mathscr{D}} \vdash \mathscr{D}$ and some G', where G_C is the complete graph whose nodes are the free process names in C. The choice of G_C is motivated by observing that (i) all top-level processes should know each other and (ii) eventual connections between processes not occuring in C do not affect its typability.

Well-typed choreographies enjoy progress, i.e., they either terminate or diverge.[5] Also, typing is preserved by reductions.

Theorem 1 (Progress and Type Preservation)**.** *Let $\langle \mathscr{D}, C \rangle$ be a procedural choreography. If $\Gamma \vdash \mathscr{D}$ and $\Gamma; G_1 \vdash C \triangleright G'_1$ for some Γ, G_1 and G'_1, then one of the following holds.*

- *$C \preceq_{\mathscr{D}} \mathbf{0}$;*
- *for every σ, there exist G_2, C' and σ' such that $G_1, C, \sigma \to_{\mathscr{D}} G_2, C', \sigma'$ and $\Gamma'; G_2 \vdash C' \triangleright G'_2$ for some $\Gamma' \supseteq \Gamma$ and G'_2.*

[5] Since we are interested in communications, we assume that evaluation of functions and expressions always terminates on values with the right types (see also § 7, Faults).

The proof of this result is included in the Appendix.

As usual, the hypothesis in Theorem 1 is not necessary: the process

$$\mathsf{p}\,\mathsf{start}\,\mathsf{q}^{\mathbb{N}}, \mathsf{r}^{\mathbb{N}}; \mathsf{if}\ \mathsf{p.false}\ \mathsf{then}\ \mathsf{q}.0\ \texttt{->}\ \mathsf{r}; \mathbf{0}\ \mathsf{else}\ \mathbf{0}$$

is not typable, as q and r cannot know each other, but that communication is never reached in actual execution. This is unavoidable, as undecidability of deadlock-freedom for PC can be established using Rice's theorem, as usual.

Checking that $\Gamma \vdash \langle \mathscr{D}, C\rangle$ is not trivial, as it requires "guessing" $\Gamma_{\mathscr{D}}$. However, this set can be computed from $\langle \mathscr{D}, C\rangle$. The key idea is that type-checking may require expanding recursive definitions, but their parameters only need to be instantiated with process names from a finite set.

Theorem 2. *Given Γ, $\mathscr{D}$ and C, $\Gamma \vdash \langle \mathscr{D}, C\rangle$ is decidable.*

Proof. We first observe that deciding whether $\Gamma; G \vdash C \triangleright G'$ is completely mechanical, as the typing rules are deterministic. Furthermore, those rules can also be used to construct G' from G and C.

We can therefore give a simple non-deterministic algorithm for deciding whether $\Gamma \vdash \langle \mathscr{D}, C\rangle$. Given Γ, $\mathscr{D}$ and C, we non-deterministically guess graphs G_i and G'_i for each procedure name X_i and set $\Gamma_{\mathscr{D}} = \{X_i(\widetilde{\mathsf{q}_i}) : G_i \triangleright G'_i$. Since the graphs G_i and G'_i have the set $\widetilde{\mathsf{q}_i}$ as vertices, there is only a finite number of possibilities. For each X_i, we then try to construct G''_i such that $\Gamma; G_i \vdash C_{X_i} \triangleright G''_i$; if this succeeds, we then remove any vertices not in $\widetilde{\mathsf{q}}$ from G''_i, and edges leading to those vertices, and check whether we obtain G'_i. If we can do this for all X_i, we have established that $\Gamma_{\mathscr{D}} \vdash \mathscr{D}$. Finally, we check that $\Gamma, \Gamma_{\mathscr{D}}; G_C \vdash C \triangleright G'$, again with G' inferred.

If $\Gamma \vdash \langle \mathscr{D}, C\rangle$, then there is one choice of $\{G_i\}_i$ and $\{G'_i\}_i$ that leads to a successful branch. Otherwise, the algorithm fails. □

Although this proof establishes decidability, the number of possibilities to test is unrealistic in practice: for a procedure X with n parameters, there are $2^{\frac{n(n-1)}{2}}$ possibilities for G_X and a similar number for G'_X, even if we restrict the search to those G'_X that contain G_x. In the Appendix, we include an alternative proof that iteratively constructs these sets in a more clever way.

Using Theorem 2, we can readily obtain a type inference algorithm for PC, as we only need to "guess" types for the processes in the choreography.

Theorem 3. *There is an algorithm that, given any $\langle \mathscr{D}, C\rangle$, outputs one of the following:*

- *a set Γ such that $\Gamma \vdash \langle \mathscr{D}, C\rangle$, if such a Γ exists;*
- `NO`*, if no such Γ exists.*

Proof. Construct Γ by going through C and adding $\mathsf{p} : T_{\mathsf{p}}$ every time there is an action that depends on p's type (i.e. p is a sender or receiver in a communication, or an argument of a procedure call). If Γ contains two different types for any

process, then output NO. Otherwise, check whether $\Gamma \vdash \langle \mathscr{D}, C \rangle$; in the negative case, output NO, else output Γ. Termination is trivial, since C is finite and type checking is decidable. This algorithm does not necessarily assign a type to all processes in C, in case C contains processes whose memory is never accessed. □

As a corollary of the proof, we can also infer the types for parameters of procedural definitions and freshly created processes.

Corollary 1. *The types of arguments in procedure definitions and the types of freshly created processes can be inferred automatically.*

Proof. Inferring the types of freshly created processes is analogous to the previous proof. For parameters of procedure definitions, define an operator $\mathscr{T}_{\mathbb{T}}$ over tuples of typing contexts (one for each X_i defined in $\mathscr{D}$) that generates a typing context for each X_i in the same way as in the previous proof. If any contradictions are found, then fail. The result of applying $\mathscr{T}_{\mathbb{T}}$ from the tuple of empty contexts does not necessarily include types for all parameters (for example, if procedure X_i has an argument p that is only passed as a parameter to another procedure X_j), so iterate $\mathscr{T}_{\mathbb{T}}$ until either failure occurs (in which case the X_is are not properly defined) or a fixpoint is reached. Finally, assign a random type (e.g. $\mathbb{N}$) to each process variable that has not received a type during this procedure. The algorithm readily extends to infer the types of processes created inside procedure definitions. □

Remark 4 (Inferring introductions)**.** *Theorems 3 and 1 allow us to omit all type annotations in choreographies, if the types of functions and expressions at processes are known (i.e., given in $\vdash_{\mathbb{T}}$). Thus, programmers can write choreographies as in our examples.*

*The same reasoning can be used to infer missing introductions (*p : q <-> r*) in a choreography automatically, thus lifting the programmer also from having to think about connections. However, while the types inferred for a choreography do not affect its behaviour, the placement of introductions does. In particular, when invoking procedures one is faced with the choice of adding the necessary introductions inside the procedure definition (weakening the conditions for its invocation) or in the code calling it (making the procedure body more efficient).*

Example 6. *Consider a procedure*

$$X(\mathsf{p},\mathsf{q},\mathsf{r}) = \mathsf{p}.*\ \texttt{->}\ \mathsf{q}; \mathsf{p} : \mathsf{q}\ \texttt{<->}\ \mathsf{r}; \mathsf{q}.*\ \texttt{->}\ \mathsf{r}$$

whose invocation requires only that p *be connected to* q *and* r*. If we invoke X twice with the same parameters, as in $X\langle \mathsf{p},\mathsf{q},\mathsf{r}\rangle; X\langle \mathsf{p},\mathsf{q},\mathsf{r}\rangle$, we end up performing the same introduction* p : q <-> r *twice. We could avoid this duplication by rewriting X as $X(\mathsf{p},\mathsf{q},\mathsf{r}) =$* p.* -> q; q.* -> r *and then performing the introduction only once before invoking the procedure –* p : q <-> r*; $X\langle \mathsf{p},\mathsf{q},\mathsf{r}\rangle; X\langle \mathsf{p},\mathsf{q},\mathsf{r}\rangle$. However, this makes invoking X more complicated. Deciding which variant is best depends heavily on the context.*

Remark 5. *Well-typed choreographies enjoy global deadlock-freedom, as shown above. However, they do not necessarily enjoy liveness: every process that is not terminated eventually reduces. For example, if procedure X is defined as $X(\mathsf{p},\mathsf{q}) = \mathsf{p}.*\ \texttt{->}\ \mathsf{q}; X\langle \mathsf{p},\mathsf{q}\rangle$, then in the choreography $X\langle \mathsf{p},\mathsf{q}\rangle; \mathsf{p}.*\ \texttt{->}\ \mathsf{r}$ process r is not terminated, but it will be forever waiting for a message from p that never arises.*

The feature of PC that is responsible for this behaviour is the ability to write calls to procedures followed by arbitrary code. If we restrict PC to those choreographies where, in all subterms of the form $I; C'$, we have $C' = \mathbf{0}$, then we gain liveness thanks to the argument: if p is not terminated in C, then the sequence of actions syntactically preceding the first action involving p (which may require unfolding procedure definitions in C) is finite, and can be reduced in finite time, since it cannot include any procedure calls.

Studying liveness for PC is an interesting topic that we leave to future work. Liveness is most useful under fair semantics, and the semantics that we presented for PC is not fair (similar to the situation in other choreography calculi). Consider a procedure definition $Y(\mathsf{p},\mathsf{q},\mathsf{r},\mathsf{s}) = \mathsf{p}.*\ \texttt{->}\ \mathsf{q}; \mathsf{r}.*\ \texttt{->}\ \mathsf{s}; Y\langle \mathsf{p},\mathsf{q},\mathsf{r},\mathsf{s}\rangle$. Due to swapping, there is an infinite sequence of reductions of $Y\langle \mathsf{p},\mathsf{q},\mathsf{r},\mathsf{s}\rangle$ consisting only of communications from p to q; thus r and s, both of which have communications enabled, can starve. This example can be written in most other choreography languages, since it only involves communications and tail recursion.

Previous works on choreographic programming with procedures have only tail recursion, so some of them would trivially enjoy liveness if they were specified with a fair semantics as well, e.g., the model by Carbone and Montesi (2013).

4. Synthesising Process Implementations

We now present our EndPoint Projection (EPP), which compiles a choreography to a concurrent implementation represented in terms of a process calculus.

4.1. Procedural Processes (PP)

We first introduce our target process model, Procedural Processes (PP), which features syntactic primitives similar to those of PC.

Syntax. A procedural network is a pair $\langle \mathscr{B}, N\rangle$, where $\mathscr{B}$ is a set of procedure definitions and N (the network) is a parallel composition of processes. A process is written as $\mathsf{p} \triangleright_v B$, where p is its name, v is its value, and B is its behaviour. The full syntax of PP is given in Figure 5. Values, expressions and functions are as in PC; again, the procedures defined in $\mathscr{B}$ may be invoked both in N and in their own definitions.

We comment on the syntax of behaviours. A process executing a send term $\mathsf{q}!e; B$ sends the evaluation of expression e to q, and proceeds as B. Term $\mathsf{p}?f; B$ is the dual receiving action: the process executing it receives a value from p, combines it with its value as specified by f, and then proceeds as B. Term $\mathsf{q}!!\mathsf{r}$

$$\mathscr{B} ::= X(\tilde{\mathsf{q}}) = B, \mathscr{B} \mid \emptyset \qquad B ::= \mathsf{q}!e; B \mid \mathsf{p}?f; B \mid \mathsf{q}!!\mathsf{r}; B \mid \mathsf{p}?\mathsf{r}; B \mid \mathsf{q} \oplus \ell; B \mid \mathsf{p}\&\{\ell_i : B_i\}_{i\in I}; B$$
$$N, M ::= \mathsf{p} \triangleright_v B \mid (N \mid M) \mid \mathbf{0} \qquad \mid \mathbf{0} \mid \mathsf{start}\, \mathsf{q}^T \triangleright B_2; B_1 \mid \mathsf{if}\, e\, \mathsf{then}\, B_1\, \mathsf{else}\, B_2; B \mid X\langle \tilde{\mathsf{p}} \rangle; B \mid \mathbf{0}; B$$

Figure 5: Procedural Processes, Syntax.

$$\frac{u = (f[w/*])(e[v/*])}{\mathsf{p} \triangleright_v \mathsf{q}!e; B_1 \mid \mathsf{q} \triangleright_w \mathsf{p}?f; B_2 \to_{\mathscr{B}} \mathsf{p} \triangleright_v B_1 \mid \mathsf{q} \triangleright_u B_2} \lfloor \mathrm{P}|\mathrm{Com} \rceil$$

$$\frac{j \in I}{\mathsf{p} \triangleright_v \mathsf{q} \oplus l_j; B \mid \mathsf{q} \triangleright_w \mathsf{p}\&\{l_i : B_i\}_{i\in I} \to_{\mathscr{B}} \mathsf{p} \triangleright_v B \mid \mathsf{q} \triangleright_w B_j} \lfloor \mathrm{P}|\mathrm{Sel} \rceil$$

$$\frac{i = 1 \text{ if } e[v/*] = \mathsf{true}, \quad i = 2 \text{ otherwise}}{\mathsf{p} \triangleright_v \mathsf{if}\, e\, \mathsf{then}\, B_1\, \mathsf{else}\, B_2 \to_{\mathscr{B}} \mathsf{p} \triangleright_v B_i} \lfloor \mathrm{P}|\mathrm{Cond} \rceil$$

$$\frac{\mathsf{q}' \text{ fresh}}{\mathsf{p} \triangleright_v (\mathsf{start}\, \mathsf{q}^T \triangleright B_2; B_1) \to_{\mathscr{B}} \mathsf{p} \triangleright_v B_1[\mathsf{q}'/\mathsf{q}] \mid \mathsf{q}' \triangleright_{\perp_T} B_2} \lfloor \mathrm{P}|\mathrm{Start} \rceil$$

$$\frac{}{\mathsf{p} \triangleright_v \mathsf{q}!!\mathsf{r}; B_1 \mid \mathsf{q} \triangleright_w \mathsf{p}?\mathsf{r}; B_2 \mid \mathsf{r} \triangleright_u \mathsf{p}?\mathsf{q}; B_3 \to_{\mathscr{B}} \mathsf{p} \triangleright_v B_1 \mid \mathsf{q} \triangleright_w B_2 \mid \mathsf{r} \triangleright_u B_3} \lfloor \mathrm{P}|\mathrm{Tell} \rceil$$

$$\frac{N \to_{\mathscr{B}} N'}{N \mid M \to_{\mathscr{B}} N' \mid M} \lfloor \mathrm{P}|\mathrm{Par} \rceil \qquad \frac{N \preceq_{\mathscr{B}} M \quad M \to_{\mathscr{B}} M' \quad M' \preceq_{\mathscr{B}} N'}{N \to_{\mathscr{B}} N'} \lfloor \mathrm{P}|\mathrm{Struct} \rceil$$

Figure 6: Procedural Processes, Semantics.

sends process name r to q and process name q to r, making q and r "aware" of each other. The dual action is $\mathsf{p}?\mathsf{r}$, which receives a process name from p that replaces the bound variable r in the continuation. Term $\mathsf{q} \oplus \ell; B$ sends the selection of a label ℓ to process q. Selections are matched with branching terms $\mathsf{p}\&\{\ell_i : B_i\}_{i\in I}$, which can receive a selection for any of the labels ℓ_i and proceed as the corresponding B_i. Branching terms must offer at least one branch. Term $\mathsf{start}\, \mathsf{q} \triangleright B_2; B_1$ starts a new process (with a fresh global name) executing B_2, and proceeds in parallel as B_1. Conditionals, procedure calls, and termination are standard. Term $\mathsf{start}\, \mathsf{q} \triangleright B_2; B_1$ binds q in B_1, and $\mathsf{p}?\mathsf{r}; B$ binds r in B.

Semantics. The transition semantics of PP formalises the intuition given above. The rules defining the reduction relation $\to_{\mathscr{B}}$ for PP are shown in Figure 6. As in PC, they are parameterised on the set of behavioural procedures $\mathscr{B}$. Rule $\lfloor \mathrm{P}|\mathrm{Com} \rceil$ models value communication: a process p executing a send action towards a process q can synchronise with a receive-from-p action at q; in the reductum, f is used to update the memory of q by combining its contents with the value sent by p. The placeholder $*$ is replaced with the current value of p in e (resp. q in f). Rule $\lfloor \mathrm{P}|\mathrm{Sel} \rceil$ is the standard rule for selection due to Honda et al. (1998), where the sender process selects one of the branches offered by the receiver.

Rule $\lfloor \mathrm{P}|\mathrm{Start} \rceil$ requires the name of the created process to be globally fresh.

$$\frac{}{\mathbf{0}; B \preceq_{\mathscr{B}} B}\ \lfloor \text{P}|\text{End} \rceil \qquad \frac{}{\mathsf{p} \triangleright_v \mathbf{0} \preceq_{\mathscr{B}} \mathbf{0}}\ \lfloor \text{P}|\text{AZero} \rceil \qquad \frac{}{N \mid \mathbf{0} \preceq_{\mathscr{B}} N}\ \lfloor \text{P}|\text{NZero} \rceil$$

$$\frac{X(\widetilde{\mathsf{q}^T}) = B_X \in \mathscr{B}}{X\langle \tilde{\mathsf{p}} \rangle; B \preceq_{\mathscr{B}} B_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \fatsemi B}\ \lfloor \text{P}|\text{Unfold} \rceil$$

Figure 7: Procedural Processes, Structural precongruence $\preceq_{\mathscr{B}}$.

Rule $\lfloor$P|Tell$\rceil$ establishes a three-way synchronisation, allowing a process to introduce two others. Since the received names are bound at the receivers, we use α-conversion to make the receivers agree on each other's name, as in session types (Honda et al., 1998). In this we follow the tradition from process calculi, and depart from the Barendregt variable convention: in the term $\mathsf{p} \triangleright_v \mathsf{q}!!\mathsf{r}; B_1 \mid \mathsf{q} \triangleright_w \mathsf{p}?\mathsf{r}; B_2 \mid \mathsf{r} \triangleright_u \mathsf{p}?\mathsf{q}; B_3$, the process name r is used both free (as an identifier and in p's behaviour) and bound (in the receive action at q).

All other rules are standard. Relation $\rightarrow_{\mathscr{B}}$ is closed under a structural precongruence $\preceq_{\mathscr{B}}$, defined by the rules in Figure 7 and associativity and commutativity of parallel ($\mid$). Rule $\lfloor$P|Unfold$\rceil$, which expands procedure calls. It uses again the $\fatsemi$ operator, defined as for PC but with terms in the PP language.

Remark 6. *The three-way synchronisation in rule $\lfloor P|\, Tell \rceil$ could be encoded with two standard two-way communications of names, as done by Sangiorgi and Walker (2001) for π-calculus (Sangiorgi and Walker, 2001) (see also Remark 2). However, our choice gives a clearer formulation of EPP.*

Example 7. *We show a process implementation of the merge sort choreography in Example 1 from § 1. All processes are annotated with type* **List**(T) *(omitted);* id *is the identity function (Example 2).*

```
MSp(p) = if is_small then 0
  else start q1 ▷ (p?id; MSp<q1>; p!*);
       start q2 ▷ (p?id; MSp<q2>; p!*);
       q1!split1; q2!split2; q1?id; q2?merge
```

In the next section, we show that our EPP generates this process implementation automatically from the choreography in Example 1.

4.2. EndPoint Projection (EPP)

We now show how to compile programs in PC to processes in PP.

Behaviour Projection. We start by defining how to project the behaviour of a single process p, a partial function denoted $[\![C]\!]_{\mathsf{p}}$. The rules defining behaviour projection are given in Figure 8. Each choreography term is projected to the local action of the process that we are projecting. For example, a communication term $\mathsf{p}.e$ -> $\mathsf{q}.f$ projects a send action for the sender p, a receive action for the receiver q, or skips to the continuation otherwise. The rules for projecting a selection or an introduction (name mobility) are similar.

$$[\![\mathsf{p}.e \texttt{ -> } \mathsf{q}.f; C]\!]_\mathsf{r} = \begin{cases} \mathsf{q}!e; [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p}?f; [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_\mathsf{r} & \text{otherwise} \end{cases} \qquad [\![\mathsf{p} \texttt{ -> } \mathsf{q}[\ell]; C]\!]_\mathsf{r} = \begin{cases} \mathsf{q} \oplus l; [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p}\&\{l : [\![C]\!]_\mathsf{r}\} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_\mathsf{r} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{p} : \mathsf{q} \texttt{ <-> } \mathsf{r}; C]\!]_\mathsf{s} = \begin{cases} \mathsf{q}!!\mathsf{r}; [\![C]\!]_\mathsf{s} & \text{if } \mathsf{s} = \mathsf{p} \\ \mathsf{p}?\mathsf{r}; [\![C]\!]_\mathsf{s} & \text{if } \mathsf{s} = \mathsf{q} \\ \mathsf{p}?\mathsf{q}; [\![C]\!]_\mathsf{s} & \text{if } \mathsf{s} = \mathsf{r} \\ [\![C]\!]_\mathsf{s} & \text{otherwise} \end{cases} \qquad [\![X\langle \tilde{\mathsf{p}} \rangle; C]\!]_\mathsf{r} = \begin{cases} X_i\langle \tilde{\mathsf{p}} \rangle; [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{p}_i \\ [\![C]\!]_\mathsf{r} & \text{otherwise} \end{cases}$$

$$[\![\mathbf{0}]\!]_\mathsf{r} = \mathbf{0} \qquad [\![\mathbf{0}; C]\!]_\mathsf{r} = [\![C]\!]_\mathsf{r}$$

$$[\![\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C]\!]_\mathsf{r} = \begin{cases} \mathsf{if}\ e\ \mathsf{then}\ [\![C_1]\!]_\mathsf{r}\ \mathsf{else}\ [\![C_2]\!]_\mathsf{r}; [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{p} \\ ([\![C_1]\!]_\mathsf{r} \sqcup [\![C_2]\!]_\mathsf{r}); [\![C]\!]_\mathsf{r} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{p}\ \mathsf{start}\ \mathsf{q}^T; C]\!]_\mathsf{r} = \begin{cases} \mathsf{start}\ \mathsf{q} \triangleright [\![C]\!]_\mathsf{q}; [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{p} \\ [\![C]\!]_\mathsf{r} & \text{otherwise} \end{cases}$$

Figure 8: Procedural Choreographies, Behaviour Projection.

The rule for projecting a conditional uses the partial merging operator $\sqcup$, which adapts the homonymous operator due to Carbone et al. (2012) to PC. Intuitively, $B \sqcup B'$ is isomorphic to B and B' up to branching, where branches of B or B' with distinct labels are also included. This is expressed by the following equation:

$$\begin{aligned}(\mathsf{p}\&\{l_i : B_i\}_{i\in I}; B) \sqcup \big(\mathsf{p}\&\{l_j : B'_j\}_{j\in J}; B'\big) = \\ \mathsf{p}\&\big(\{l_k : (B_k \sqcup B'_k)\}_{k\in I\cap J} \cup \{l_i : B_i\}_{i\in I\setminus J} \cup \{l_j : B'_j\}_{j\in J\setminus I}\big); (B \sqcup B')\end{aligned}$$

For all other syntactic constructs, merging proceeds homomorphically, e.g.:

$$(\mathsf{q}!e; B) \sqcup (\mathsf{q}!e; B') = \mathsf{q}!e; (B \sqcup B')$$
$$(\mathsf{start}\ \mathsf{q} \triangleright B_2; B_1) \sqcup (\mathsf{start}\ \mathsf{q} \triangleright B'_2; B'_1) = \mathsf{start}\ \mathsf{q} \triangleright (B_2 \sqcup B'_2)\,; (B_1 \sqcup B'_1)$$

and $B \sqcup B'$ is undefined if B and B' do not agree on their initial action. Merging allows the process that decides a conditional to inform other processes of its choice later on, using selections, and it is found repeatedly in choreography models, e.g., those by Lanese et al. (2008), Carbone et al. (2012), Coppo et al. (2016) and Cruz-Filipe and Montesi (2016a).

Building on behaviour projection, we define how to project the set $\mathscr{D}$ of procedure definitions. We need to consider two main aspects. The first is that, at runtime, the choreography may invoke a procedure X multiple times, but potentially passing a process r at different argument positions each time. This means that r may be called to play different "roles" in the implementation of the procedure. For this reason, we project the behaviour of each possible process parameter p as the local procedure X_p. The second aspect is: depending on the role that r is called to play by the choreography, it needs to know the names of

the other processes that it is supposed to communicate with in the choreographic procedure. We deal with this by simply passing all arguments (some of which may be unknown to the process invoking the procedure). This is not a problem: for typable choreographies, typing ensures that processes unknown to e.g. p do not occur in the projected procedure X_{p} (so they act as "dummies").[6]

We thus define

$$[\![\mathscr{D}]\!] = \bigcup \left\{ [\![X(\widetilde{\mathsf{q}^T}) = C]\!] \mid X(\widetilde{\mathsf{q}^T}) = C \in \mathscr{D} \right\}$$

where, for $\widetilde{\mathsf{q}^T} = \mathsf{q}_1^{T_1}, \ldots, \mathsf{q}_n^{T_n}$, we set

$$[\![X(\widetilde{\mathsf{q}^T}) = C]\!] = \{X_1(\tilde{\mathsf{q}}) = [\![C]\!]_{\mathsf{q}_1}, \ldots, X_n(\tilde{\mathsf{q}}) = [\![C]\!]_{\mathsf{q}_n}\}$$

Definition 1 (EPP)**.** *Given a procedural choreography $\langle \mathscr{D}, C \rangle$ and a state σ, the EPP $[\![\mathscr{D}, C, \sigma]\!]$ is the parallel composition of the processes in C with all definitions from $\mathscr{D}$:*

$$[\![\mathscr{D}, C, \sigma]\!] = \langle [\![\mathscr{D}]\!], [\![C, \sigma]\!] \rangle = \left\langle [\![\mathscr{D}]\!], \prod_{\mathsf{p} \in \mathsf{pn}(C)} \mathsf{p} \triangleright_{\sigma(\mathsf{p})} [\![C]\!]_{\mathsf{p}} \right\rangle$$

where $[\![C, \sigma]\!]$, the EPP of C wrt state σ, is independent of $\mathscr{D}$.

Since every σ is total, if $[\![C, \sigma]\!]$ is defined for some σ, then $[\![C, \sigma']\!]$ is defined also for all other σ'; in this case, we say that C is *projectable*. The same holds for $[\![\mathscr{D}, C, \sigma]\!]$.

Example 8. *The EPP of the choreography in Example 1 is given in Example 7.*

Example 9. *For an example involving merging and introductions, we project the procedure* `par_download` *(Example 5) for process* `s`*, omitting type annotations.*

```
par_download_s(c,s) = c&{
  more: start s' ▷ (s?c; c?id; c?c'; download_s<c',s'>);
        c!!s'; par_download_s<c,s>
  end: 0
}
```

Observe that we invoke procedure $\texttt{download}_{\texttt{s}}$*, since* `s'` *occurs in the position of* `download`*'s formal argument* `s`*.*

4.3. Properties of EPP

EPP guarantees the following operational correspondence, which is the hallmark correctness-by-construction property of choreography languages. It uses a

[6] We do this for clarity, since it yields a simpler formulation of EPP. In practice, we can annotate the EPP by analysing which parameters of each recursive definition are actually used in each of its projections, and instantiating only those – see Remark 7 below. Likewise, in a realistic setting a process should not pass its own name as an argument to a procedure call, but rather a special keyword such as `this`.

pruning relation $\prec$, also introduced by Carbone et al. (2012), which eliminates branches introduced by the merging operator $\sqcup$ when they are not needed anymore to follow the originating choreography. We write $N \succ N'$ for $N' \prec N$. Pruning does not alter reductions: Carbone et al. (2012) also showed that the eliminated branches are never selected.

Theorem 4 (EPP Theorem)**.** *If $\langle \mathscr{D}, C\rangle$ is projectable, G is a connection graph, and there exist Γ and G^* such that $\Gamma \vdash \mathscr{D}$ and $\Gamma; G \vdash C \triangleright G^*$, then, for all σ, the following hold.*

Completeness. *If $G, C, \sigma \to_{\mathscr{D}} G', C', \sigma'$, then $[\![C, \sigma]\!] \to_{[\![\mathscr{D}]\!]} \succ [\![C', \sigma']\!]$.*

Soundness. *If $[\![C, \sigma]\!] \to_{[\![\mathscr{D}]\!]} N$, then $G, C, \sigma \to_{\mathscr{D}} G', C', \sigma'$ for some G', C' and σ' such that $[\![C', \sigma']\!] \prec N$.*

Proof sketch (Theorem 4). The structure of the proof follows those of Carbone et al. (2012), Carbone and Montesi (2013) and Montesi (2013), so we only show the most interesting differing details. In particular, we need to be careful about how we deal with connections, which is a new key ingredient in PC wrt previous work. We demonstrate this point for the direction of *(Completeness)*; the direction for *(Soundness)* is adapted similarly. The proof proceeds by induction on the derivation of $G, C, \sigma \to_{\mathscr{D}} G'', C', \sigma'$. The interesting cases are reported below.

- $\lfloor$C|Tell$\rceil$: From the definition of EPP we get:

  $$[\![\mathsf{p} : \mathsf{q} \texttt{<->} \mathsf{r}; C^\circ, \sigma]\!] \preceq \mathsf{p} \triangleright_{\sigma(\mathsf{p})} \mathsf{q}!!\mathsf{r}; [\![C^\circ]\!]_{\mathsf{p}} \mid \mathsf{q} \triangleright_{\sigma(\mathsf{q})} \mathsf{p}?\mathsf{r}; [\![C^\circ]\!]_{\mathsf{q}} \mid \mathsf{r} \triangleright_{\sigma(\mathsf{r})} \mathsf{p}?\mathsf{q}; [\![C^\circ]\!]_{\mathsf{r}} \mid N$$

  By $\lfloor$P|Tell$\rceil$ we get:

  $$[\![\mathsf{p} : \mathsf{q} \texttt{<->} \mathsf{r}; C^\circ, \sigma]\!] \to \mathsf{p} \triangleright_{\sigma'(\mathsf{p})} [\![C^\circ]\!]_{\mathsf{p}} \mid \mathsf{q} \triangleright_{\sigma'(\mathsf{q})} [\![C^\circ]\!]_{\mathsf{q}} \mid \mathsf{r} \triangleright_{\sigma'(\mathsf{r})} [\![C^\circ]\!]_{\mathsf{r}} \mid N$$

  which proves the thesis, since the projection of C' remains unchanged for the other processes (N stays the same).

- $\lfloor$C|Start$\rceil$: This is the most interesting case. From the definition of EPP we get:

  $$[\![\mathsf{p}\ \mathsf{start}\ \mathsf{q}^T; C^\circ, \sigma]\!] \preceq \mathsf{p} \triangleright_{\sigma(\mathsf{p})} \mathsf{start}\ \mathsf{q}^T \triangleright [\![C^\circ]\!]_{\mathsf{q}}; [\![C^\circ]\!]_{\mathsf{p}} \mid N$$

  From the semantics of PP we get, for some fresh q':

  $$[\![\mathsf{p}\ \mathsf{start}\ \mathsf{q}^T; C^\circ, \sigma]\!] \to \mathsf{p} \triangleright_{\sigma'(\mathsf{p})} ([\![C^\circ]\!]_{\mathsf{p}})[\mathsf{q}'/\mathsf{q}] \mid \mathsf{q}' \triangleright_{\bot_T} [\![C^\circ]\!]_{\mathsf{q}'} \mid N$$

  Since q' does not occur in C, C is α-equivalent to $\mathsf{p}\ \mathsf{start}\ \mathsf{q}'^T; (C^\circ[\mathsf{q}'/\mathsf{q}])$. We now have to prove that:

  $$[\![C^\circ[\mathsf{q}'/\mathsf{q}], \sigma]\!] \preceq \mathsf{p} \triangleright_{\sigma'(\mathsf{p})} ([\![C^\circ]\!]_{\mathsf{p}})[\mathsf{q}'/\mathsf{q}] \mid \mathsf{q}' \triangleright_{\bot_T} [\![C^\circ]\!]_{\mathsf{q}'} \mid N$$

  We observe that this is true only if process q does not occur free in N, i.e., q may appear in N only inside the scope of a binder. Such a binder must be of the form $\mathsf{r}?\mathsf{q}; B$. This is guaranteed by the fact that C is well-typed, since the typing rules prevent other processes in N to communicate with q without being first introduced. □

By combining Theorem 4 with Theorem 1, we conclude that the projections of typable PC terms never deadlock.

Corollary 2 (EPP Progress)**.** *Let N be a network such that $N = [\![C, \sigma]\!]$ for some C and σ, and assume that there exist Γ, G and G' such that $\Gamma; G \vdash C \triangleright G'$ and $\Gamma \vdash \mathscr{D}$. Then one of the following holds.*

- *$N \preceq_{[\![\mathscr{D}]\!]} \mathbf{0}$ (N has terminated);*
- *there exists N' such that $N \to_{[\![\mathscr{D}]\!]} N'$ (N can reduce).*

Remark 7 (Projections of procedure definitions)**.** *We shortly discuss how to use typing to refine the projection of procedures, so that no process includes references to uninstantiated process variables. The key idea is to use typing information to decide which parameters should be kept in each projection. In other words, when computing $[\![X(\widetilde{\mathsf{q}^T}) = C]\!]$, the projected procedure X_i only contains as arguments those q_j such that $\mathsf{q}_i \overset{G_X}{\longleftrightarrow} \mathsf{q}_j$, where G_X is given by typing. (This definition is non-deterministic, but it can be made deterministic e.g. by using the minimal graph G_X computed by the type inference algorithm.) A simple annotation can then ensure that projected procedure calls only keep the arguments in the corresponding positions, and typing guarantees that all those arguments are known at runtime by the process invoking the recursive call. In § 6.1 we show an implementation of Quicksort showcasing this reasoning.*

Remark 8 (Amendment)**.** *A choreography can only be unprojectable because of unmergeable subterms, and thus can be made projectable by adding label selections. This can be formalised in an amendment algorithm, similar to the ones of Lanese et al. (2013) and Cruz-Filipe and Montesi (2016a). For example, the first (unprojectable) choreography in Remark 3 can be amended to the projectable choreography presented at the end of the same remark.*

The same argument as in Remark 4 applies: amendment allows us to disregard label selections, but placing them manually can be useful. For example, suppose p *makes a choice that affects* q *and* r*. If* q *has to perform a slower computation as a result, then it makes sense for* p *to notify* q *first.*

5. Parameter Lists

Before we present more complex examples of choreographies that we can write in PC, we describe a simple extension to its syntax that enhances its expressivity. This extension consists of allowing procedure parameters to be lists of processes, upon which procedures can act uniformly by recursion. Informally, a procedure parameter can now be either a process or a list of processes all with the same type. Lists of processes can only be used as the arguments of procedure calls; however, they may be manipulated by means of pure total linear functions that take a list as their only argument.

Extending PC. Formally, we extend the syntax of PC as follows.

Definition 2. *An* acceptable (process) list manipulating function *is a total function f whose arguments are processes and lists of processes and whose result is either:*

- *a single process contained in one of its arguments;*
- *a list built from the processes in its arguments, which does not contain duplicates if no process occurs twice among the arguments of f.*

We assume a fixed set of acceptable list manipulating functions whose semantics is rigidly defined. Typically, these are standard functions taking the head or tail of a list, the element at a given position in a list, appending an element to a list, filtering the elements of a list according to a predicate, etc. We do not allow acceptable list manipulating functions to be nested, i.e., they can only be applied to processes or lists of processes that are explicitly defined.

In procedure definitions, we allow some arguments to be lists of uniformly typed parameters. In procedure calls, we allow some arguments to be applications of acceptable list manipulating functions, with the proviso that the results must not share processes.[7] Observe that communication actions are still only allowed to use processes directly identified by their name. For example, if procedure X takes a list P as an argument, then X's body may call other procedures on P, $\mathsf{head}(P)$ or $\mathsf{tail}(P)$, but it may not include a communication action involving $\mathsf{head}(P)$.

The semantics of PC is extended with the new garbage collection rule $X\langle\langle\rangle\rangle \preceq_{\mathscr{D}} \mathbf{0}$, stating that calling a procedure with an empty list yields a terminated choreography ($\mathbf{0}$). In order to type procedures that take process lists as arguments, in a typing $X : G \triangleright G'$ we allow the vertices of G and G' to be not only processes, but also (formal) lists. This requires adjusting the premise of $\lfloor \mathrm{T} | \mathrm{Call} \rceil$ to the case when some of the arguments are lists. When a concrete list is given as argument, we update the premise $G_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \subseteq G$ to check that: if G_X contains an edge between two process lists P and Q and these lists are instantiated by $\tilde{\mathsf{p}}$ and $\tilde{\mathsf{q}}$, respectively, then G must contain edges between p and q for every $\mathsf{p} \in \tilde{\mathsf{p}}$ and every $\mathsf{q} \in \tilde{\mathsf{q}}$. The interpretation of an edge between an argument process and an argument list is similar. In the case where the instantiated process or list is the result of a function call, we test all arguments to the function.[8]

[7] Our type system assumes that this condition holds, which must be ensured by the programmer when writing procedure calls. Since acceptable list manipulating functions cannot be nested and their semantics is fixed, the programmer should easily be able to guarantee this condition.

[8] This may sound very restrictive, but in practice it is quite reasonable as long as the functions do not take spurious arguments. Consider, e.g., $f(\mathsf{p}, Q)$. The result of this function is likely a list containing p and some elements from Q. If we need to check that, e.g., $f(\mathsf{p}, Q)$ is connected to q, then we are requiring that p is connected to q and that all elements of Q are connected to q – and the latter condition is a test on the single edge between Q and q, since Q is uninstantiated.

To ensure projectability, we need to be able to broadcast label selections to all elements of a parameter list in a way that is recognized by the merge operator. At the choreography level, we assume that, for every label ℓ, the set $\mathscr{D}$ contains procedure definitions

$$\begin{aligned}\mathsf{gsel}_\ell(\mathsf{p}, Q) &= \mathsf{gsel}^{\mathsf{one}}_\ell(\mathsf{p}, \mathsf{hd}(Q)); \mathsf{gsel}_\ell(\mathsf{p}, \mathsf{tl}(Q))\\ \mathsf{gsel}^{\mathsf{one}}_\ell(\mathsf{p}, \mathsf{q}) &= \mathsf{p} \texttt{ -> } \mathsf{q}[\ell]\end{aligned}$$

where hd and tl are the usual head and tail functions on lists. Our syntactic restrictions on parameter lists forbid us from writing a communication action $\mathsf{p} \texttt{ -> } \mathsf{hd}(Q)[\ell]$, which is why we need the auxiliary procedure $\mathsf{gsel}^{\mathsf{one}}_\ell$.

Example 10. *We illustrate this extended language with a very simple toy choreography. Consider the procedure*

```
X(p,Q) = if p.coinflip then gsel_ok(p,Q); X(head(Q), tail(Q))
                       else gsel_ko(p,Q); 0
```

A possible reduction path for $\texttt{X(p,}\langle \mathsf{q}_1,\mathsf{q}_2,\mathsf{q}_3\rangle\texttt{)}$*, assuming all coinflips evaluate to true, is:*

```
X(p,⟨q1,q2,q3⟩)
≼ if p.coinflip then gsel_ok(p,⟨q1,q2,q3⟩); X(q1,⟨q2,q3⟩)
                else gsel_ko(p,⟨q1,q2,q3⟩); 0
→ gsel_ok(p,⟨q1,q2,q3⟩); X(q1,⟨q2,q3⟩)
≼ gsel^one_ok(p,q1); gsel^one_ok(p,q2); gsel^one_ok(p,q3); X(q1,⟨q2,q3⟩)
≼ p->q1[ok]; p->q2[ok]; p->q3[ok]; X(q1,⟨q2,q3⟩)
→* X(q1,⟨q2,q3⟩)
≼ if q1.coinflip then gsel_ok(q1,⟨q2,q3⟩); X(q2,⟨q3⟩)
                else gsel_ko(q1,⟨q2,q3⟩); 0
→ gsel_ok(q1,⟨q2,q3⟩); X(q2,⟨q3⟩)
≼ gsel^one_ok(q1,q2); gsel^one_ok(q1,q3); X(q2,⟨q3⟩)
≼ q1->q2[ok]; q1->q3[ok]; X(q2,⟨q3⟩)
→* X(q2,⟨q3⟩)
≼ if q2.coinflip then gsel_ok(q2,⟨q3⟩); X(q3,⟨⟩)
                else gsel_ko(q2,⟨q3⟩); 0
→ gsel_ok(q2,⟨q3⟩); 0
≼ gsel^one_ok(q2,q3)
≼ q2->q3[ok]
→ 0
```

Extending PP and EPP. In order to be able to project our enriched choreographies, we need to extend the syntax of PP accordingly. Thus, we allow procedure definitions and invocations to contain parameter lists with the same restrictions as described above. In the semantics, we add the rule $\mathsf{q} \triangleright_v X_i\langle \tilde{\mathsf{p}_1}, \ldots, \tilde{\mathsf{p}_n}\rangle \preceq_{\mathscr{D}} \mathsf{q} \triangleright_v \mathbf{0}$ if $\mathsf{q} \notin \tilde{\mathsf{p}_i}$.

Furthermore, PP needs to contain terms corresponding to the projection of the procedures gsel_ℓ. We extend the set of labels with terms $\ell(Q)$ where ℓ is a label and Q a process list (possibly containing variables). These terms may then be used inside branching terms. Structural precongruence is extended with the

rule

$$\frac{\text{all } \tilde{\mathsf{r}}_j \text{ instantiated} \quad J' = \{j \in J \mid \mathsf{q} \in f_j(\tilde{\mathsf{r}}_j)\}}{\mathsf{p} \triangleright_v \mathsf{q}\&\big(\{\ell_i : B_i\}_{i\in I} \cup \{\ell_j(f_j(\tilde{\mathsf{r}}_j)) : B'_j\}_{j\in J}\big); B \preceq_{\mathscr{B}} \mathsf{p} \triangleright_v \mathsf{q}\&\big(\{\ell_i : B_i\}_{i\in I} \cup \{\ell_j : B'_j\}_{j\in J'}\big); B} \ \lfloor \text{P}|\text{EmptyBranch}\rceil$$

which replaces each $\ell(Q)$ with either ℓ (if $\mathsf{q} \in Q$) or $\mathbf{0}$ (otherwise) once Q has been instantiated.

In EPP, we need to make two changes. First, since the same parameter list may occur in several argument positions in a procedure call (for example, in $X(\mathsf{hd}(Q), \mathsf{tl}(Q))$), we need to define $[\![X\langle\tilde{\mathsf{p}}\rangle; C]\!]_Q$ as the sequential composition $X_{i_1}\langle\tilde{\mathsf{p}}_1\rangle; \ldots; X_{i_n}\langle\tilde{\mathsf{p}}_n\rangle; [\![C]\!]_Q$, where $i_1, \ldots, i_n$ are the argument positions where Q occurs. At runtime, all these calls reduce to $\mathbf{0}$ except for at most one.

Second, we need to project gsel_ℓ to the extended labels defined above by means of the rule

$$[\![\mathsf{gsel}_\ell(\mathsf{p}, Q); C]\!]_Q = \mathsf{p}\&\{\ell(Q) : [\![C]\!]_Q\}\,.$$

The merge operator then deals with extended labels as before.

Example 11. *The choreography in Example 10 projects to the following two definitions.*

```
X₁(p,Q) = if coinflip then gsel^p_ok(p,Q) else gsel^p_ko(p,Q)

X₂(p,Q) = p&{ok(Q) : X₁(head(Q),tail(Q)); X₂(head(Q),tail(Q)),
              ko(Q) : 0}
```

The previous reduction path, at the process level, begins as follows.

```
  p ▷ X₁(p,⟨q₁,q₂,q₃⟩)
| q₁ ▷ X₂(p,⟨q₁,q₂,q₃⟩)
| q₂ ▷ X₂(p,⟨q₁,q₂,q₃⟩)
| q₃ ▷ X₂(p,⟨q₁,q₂,q₃⟩)
≼
  p ▷ if coinflip then gsel^p_ok(p,⟨q₁,q₂,q₃⟩) else gsel^p_ko(p,⟨q₁,q₂,q₃⟩)
| q₁ ▷ p&{ok(⟨q₁,q₂,q₃⟩) : X₁(⟨q₁⟩,⟨q₂,q₃⟩); X₂(⟨q₁⟩,⟨q₂,q₃⟩),
          ko(⟨q₁,q₂,q₃⟩) : 0}
| q₂ ▷ p&{ok(⟨q₁,q₂,q₃⟩) : X₁(⟨q₁⟩,⟨q₂,q₃⟩); X₂(⟨q₁⟩,⟨q₂,q₃⟩),
          ko(⟨q₁,q₂,q₃⟩) : 0}
| q₃ ▷ p&{ok(⟨q₁,q₂,q₃⟩) : X₁(⟨q₁⟩,⟨q₂,q₃⟩); X₂(⟨q₁⟩,⟨q₂,q₃⟩),
          ko(⟨q₁,q₂,q₃⟩) : 0}
→
  p ▷ gsel^p_ok(p,⟨q₁,q₂,q₃⟩)
| q₁ ▷ p&{ok(⟨q₁,q₂,q₃⟩) : X₁(⟨q₁⟩,⟨q₂,q₃⟩); X₂(⟨q₁⟩,⟨q₂,q₃⟩),
          ko(⟨q₁,q₂,q₃⟩) : 0}
| q₂ ▷ p&{ok(⟨q₁,q₂,q₃⟩) : X₁(⟨q₁⟩,⟨q₂,q₃⟩); X₂(⟨q₁⟩,⟨q₂,q₃⟩),
          ko(⟨q₁,q₂,q₃⟩) : 0}
| q₃ ▷ p&{ok(⟨q₁,q₂,q₃⟩) : X₁(⟨q₁⟩,⟨q₂,q₃⟩); X₂(⟨q₁⟩,⟨q₂,q₃⟩),
          ko(⟨q₁,q₂,q₃⟩) : 0}
≽
  p ▷ q₁⊕ok; q₂⊕ok; q₃⊕ok
| q₁ ▷ p&{ok : X₁(⟨q₁⟩,⟨q₂,q₃⟩); X₂(⟨q₁⟩,⟨q₂,q₃⟩), ko : 0}
```

```
| q2 ▷ p&{ok : X1(⟨q1⟩,⟨q2,q3⟩); X2(⟨q1⟩,⟨q2,q3⟩), ko : 0}
| q3 ▷ p&{ok : X1(⟨q1⟩,⟨q2,q3⟩); X2(⟨q1⟩,⟨q2,q3⟩), ko : 0}
→
  q1 ▷ X1(⟨q1⟩,⟨q2,q3⟩); X2(⟨q1⟩,⟨q2,q3⟩)
| q2 ▷ X1(⟨q1⟩,⟨q2,q3⟩); X2(⟨q1⟩,⟨q2,q3⟩)
| q3 ▷ X1(⟨q1⟩,⟨q2,q3⟩); X2(⟨q1⟩,⟨q2,q3⟩)
≍
  q1 ▷ X1(⟨q1⟩,⟨q2,q3⟩)
| q2 ▷ X2(⟨q1⟩,⟨q2,q3⟩)
| q3 ▷ X2(⟨q1⟩,⟨q2,q3⟩)
```

Properties. Provided that our assumptions regarding procedure calls are respected, all the results that we proved for PC still hold in this extended setting with parameter lists, in particular Theorems 1 and 4. The changes to the syntax regarded arguments of procedure calls and adding the possibility of broadcasting label selections. The extensions of EPP and of the merge operator ensure that Theorem 4 still holds for the case of label selections. Regarding procedure calls, there are no changes to the structure of the proof, since the extended syntax, rules for unfolding, and garbage collection behave in the same way at the levels of choreographies and processes.

6. Examples

We now illustrate the expressivity of PC and the power of out-of-order execution by means of three examples.

6.1. QuickSort

We illustrate PC's capability of supporting the programming of divide-and-conquer algorithms by providing a detailed implementation of (concurrent) Quicksort. We begin by defining procedure `split`, which splits the (non-empty) list stored at `p` among three processes: $\mathtt{q}_<$, $\mathtt{q}_=$ and $\mathtt{q}_>$. Before giving the code for `split`, we describe the (standard) auxiliary functions and procedures that we are going to use.

We assume that all processes store objects of type `List(T)`, where `T` is some type. We also assume that these lists are implemented such that the following operations are supported and take constant time: accessing the first element (`fst`); accessing the second element (`snd`); checking that the length of a list is at most 1 (`short`); appending an element (`add`); and, appending another list (`append`). This can be readily achieved, for example, by an implementation of linked lists with pointers to the first, second and last node (and `short` simply checks where the pointer to the second node is null). We use the predicates `fst<snd` and `fst>snd` to test whether the first element of the list at a process is, respectively, smaller or greater than the second element. Finally, the procedure `pop2` (which we omit) removes the second element from the list at its argument process.

We use the abbreviation `p -> q1,...,qn[l]` to signify that `p` sends the label `l` to the processes $\mathtt{q}_1,\ldots,\mathtt{qn}$, i.e., as an abbreviation for the sequence of selections `p -> q1[l]; ...; p -> qn[l]`.

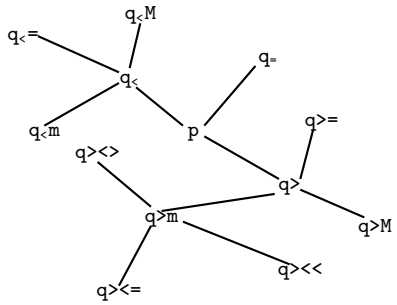

Figure 9: Example of a network connection graph after some recursive calls of `QS`.

```
split(p, q< , q= , q> ) =
  if p.short then (p -> q< , q= , q> [stop]; p.fst -> q= .add)
  else if p.fst<snd then (p -> q< [get]; p.snd -> q< .add; p -> q= , q> [skip])
       else if p.fst>snd then (p -> q> [get]; p.snd -> q> .add; p -> q< , q= [skip])
            else (p -> q= [get]; p.snd -> q= .add; p -> q< , q> [skip])
       ; pop2<p>; split<p, q< , q= , q> >
```

Procedure `split` starts by testing whether the list at process `p` is `short`. If so, its element is stored in process $q_=$ and the procedure terminates. Otherwise, we test whether the second element in the list is smaller, greater, or equal to the first element in the list, and add it to the respective process $q_<$, $q_=$, or $q_>$; then, we pop the second element of the list at `p` and recursively invoke `split`. When `split` terminates, we know that all elements in $q_<$ and $q_>$ are respectively smaller and greater than those in $q_=$.

Sending the label `skip` to the processes that will not receive messages in an iteration is required for projectability. Using `split` we can implement a robust version of Quicksort (in the sense that it works with lists containing duplicates), the procedure `QS` below. We use `p start` q_1`, ..., qn` as a shortcut for the sequence `p start` q_1`; ...; p start qn`. Observe that `split` is only called when `p` stores a non-empty list.

```
QS(p) = if p.short then 0
        else p.start q< , q= , q> ; split<p, q< , q= , q> >; QS<q< >; QS<q> >;
             q< .c -> p.id; q= .c -> p.append; q> .c -> p.append
```

Procedure `QS` implements Quicksort using its standard recursive structure. Since the created processes $q_<$, $q_=$ and $q_>$ do not have references to each other, they cannot exchange messages, and thus the recursive calls run completely in parallel. Therefore, the network of processes becomes tree-like, as exemplified in Figure 9.

Applying EPP, we get the following process procedures, where the projections of procedure definitions include only the arguments that are actually used inside the procedures (see Remark 7).

```
split_p(p, q< , q= , q> ) =
  if short then q<⊕stop; q=⊕stop; q>⊕stop; q=!fst
  else if fst<snd then q<⊕get; q<!snd; q=⊕skip; q>⊕skip
       else if fst>snd then q>⊕get; q>!snd; q<⊕skip; q=⊕skip
            else q=⊕get; q=!snd; q<⊕skip; q>⊕skip
```

```
       ; pop2<p>; split_p<p,q_<,q_=,q_>>

split_q_<(p,q) = p&{stop: 0,
                    get: p?add;split_q_<(p,q),
                    skip: split_q_<(p,q)}

QS_p(p) = if small then 0
          else (start q_< ▷ split_q_<p,q_<>; QS_p<q_<>; p!c);
               (start q_= ▷ split_q_=<p,q_=>; p!c);
               (start q_> ▷ split_q_><p,q_>>; QS_p<q_>>; p!c);
               q_<?id; q_=?append; q_>?append
```

Observe that procedure `split_q<` only takes two of the four original parameters, as the remaining ones are never used in its body.

Remark 9. *This implementation of* `split` *is suitable in a context where communication is cheap, e.g., as in object-oriented programming and/or a multithreaded application. In architectures where communications are costly, it could be better to use a* `select` *function at* `p` *to compute the lists of elements smaller than, equal to, or larger than the pivot and send each of these in a single message to* $q_<$, $q_=$ *or* $q_>$, *respectively.*

6.2. Gaussian Elimination

We now show how we can program the distributed resolution of systems of linear equations by Gaussian elimination. Let $A\vec{x} = \vec{b}$ be a system of linear equations in matrix form. We define a procedure `gauss` that applies Gaussian elimination to transform it into an equivalent system $U\vec{x} = \vec{y}$, with U upper triangular (so this system can be solved by direct substitution). We use parameter processes $\mathtt{a}_{ij}$, with $1 \leq i \leq n$ and $1 \leq j \leq n+1$. For $1 \leq i, j \leq n$, $\mathtt{a}_{ij}$ stores one value from the coefficient matrix; $\mathtt{a}_{i,n+1}$ stores the independent term in one equation. (Including the independent terms in the coefficient matrix substantially simplifies the notation, as Gaussian elimination treats the independent vector exactly as the columns in the coefficient matrix.) After execution, each $\mathtt{a}_{ij}$ stores the corresponding term in the new system. For simplicity, we assume that the matrix A is non-singular and numerically stable.

The definition of `gauss` assumes the following functions, all of which can easily be seen to be acceptable list manipulating functions: `hd` and `tl` (computing the head and tail of a list of processes); `fst` and `rest` (taking a list of processes representing a matrix and returning the first row of the matrix, or the matrix without its first row); and `minor` (removing the first row and the first column from a matrix). Processes use standard arithmetic operations to combine their value with values received.

The code of procedure `gauss` follows.

```
gauss(A) = solve(fst(A)); elim(fst(A),rest(A)); gauss(minor(A))

solve(A) = divide_all(hd(A),tl(A)); set_1(hd(A))

divideAll(a,A) = divide(a,hd(A)); divideAll(a,tl(A))
divide(a,b)    = a.* -> b.div
```

```
elim(A,B)     = elimRow(A,fst(B)); elim(A,rest(B))
elimRow(A,B)   = elimAll(tl(A),hd(B),tl(B)); set0(hd(B))
elimAll(A,m,B) = elimOne(hd(A),m,hd(B)); elimAll(tl(A),m,tl(B))
elimOne(a,m,b) = b start x; b: x<->a; b: x<->m;
                 a.* -> x.id; m.* -> x.mult; x.* -> b.minus

set0(a) = a start p; p.0 -> a.id
set1(a) = a start p; p.1 -> a.id
```

This implementation follows the standard sequential algorithm for Gaussian elimination, as described in, e.g., Algorithm 8.4 in (Grama et al., 2003). Procedure `solve` divides the first equation by the pivot, obtaining the new first equation in the reduced system. Then, `elim` uses this row to perform an elimination step, setting the first column of the coefficient matrix to zeroes. The auxiliary procedure `elimRow` performs this step at the row level, using `elimAll` to iterate through a single row and `elimOne` to perform the actual computations. The first row and the first column of the matrix are then removed in the recursive call, as they do not change further.

Despite being written as a sequential algorithm, this implementation runs concurrently, due to the implicit parallelism in the semantics of choreographies. We illustrate this behaviour by means of a concrete example. Let A be a 3×3 matrix, so there are 12 processes in total. For legibility, we write $\mathtt{b}_1$ for the independent term `a14` etc.; $\mathtt{A}=\langle \mathtt{a}_{11},\mathtt{a}_{12},\mathtt{a}_{13},\mathtt{b}_1,\mathtt{a}_{21},\mathtt{a}_{22},\mathtt{a}_{23},\mathtt{b}_2,\mathtt{a}_{31},\mathtt{a}_{32},\mathtt{a}_{33},\mathtt{b}_3\rangle$ for the matrix; $\mathtt{A}_1=\langle \mathtt{a}_{11},\mathtt{a}_{12},\mathtt{a}_{13},\mathtt{b}_1\rangle$ for the first row (likewise for $\mathtt{A}_2$ and $\mathtt{A}_3$); and $\mathtt{A'}_2=\langle \mathtt{a}_{22},\mathtt{a}_{23},\mathtt{b}_2\rangle$ and likewise for $\mathtt{A'}_3$.

Calling `gauss(A)` unfolds to

$$\mathtt{solve}(\mathtt{A}_1);\ \mathtt{elim}(\mathtt{A}_1,\langle \mathtt{A}_2,\mathtt{A}_3\rangle);\ \mathtt{solve}(\mathtt{A'}_2);\ \mathtt{elim}(\mathtt{A'}_2,\mathtt{A'}_3);\ \mathtt{solve}(\langle \mathtt{a}_{33},\mathtt{b}_3\rangle)$$

or, unfolding `elim`,

$$\mathtt{solve}(\mathtt{A}_1);\ \mathtt{elimRow}(\mathtt{A}_1,\mathtt{A}_2);\ \mathtt{elimRow}(\mathtt{A}_1,\mathtt{A}_3);\ \mathtt{solve}(\mathtt{A'}_2);\ \mathtt{elimRow}(\mathtt{A'}_2,\mathtt{A'}_3);\ \mathtt{solve}(\langle \mathtt{a}_{33},\mathtt{b}_3\rangle)$$

Unfolding $\mathtt{solve}(\mathtt{A}_1)$ is straightforward, leading to

$$\mathtt{a}_{11}.* \to \mathtt{a}_{12}.\mathtt{div};\ \mathtt{a}_{11}.* \to \mathtt{a}_{13}.\mathtt{div};\ \mathtt{a}_{11}.* \to \mathtt{b}_1.\mathtt{div};\ \mathtt{a}_{11}\ \mathtt{start}\ \mathtt{x}_1;\ \mathtt{x}_1.1 \to \mathtt{a}_{11}.\mathtt{id}$$

and likewise for the remaining calls to `solve`. In turn, $\mathtt{elimRow}(\mathtt{A}_1,\mathtt{A}_2)$ becomes

$$\mathtt{elimAll}(\langle \mathtt{a}_{12},\mathtt{a}_{13},\mathtt{b}_1\rangle,\mathtt{a}_{21},\langle \mathtt{a}_{22},\mathtt{a}_{23},\mathtt{b}_2\rangle);\ \mathtt{set}_0(\mathtt{a}_{21})$$

which again expands to

$$\mathtt{elimOne}(\mathtt{a}_{12},\mathtt{a}_{21},\mathtt{a}_{22});\ \mathtt{elimOne}(\mathtt{a}_{13},\mathtt{a}_{21},\mathtt{a}_{23});\ \mathtt{elimOne}(\mathtt{b}_1,\mathtt{a}_{21},\mathtt{b}_2);\ \mathtt{set}_0(\mathtt{a}_{21})$$

and we note that each of these procedure calls involves only communication between the processes explicitly given as arguments.

Since all these procedures involve $\mathtt{a}_{21}$, the semantics of choreographies requires them to be executed in this order. Likewise, the call to $\mathtt{elimRow}(\mathtt{A}_1,\mathtt{A}_3)$ must be executed afterwards (since it also involves processes $\mathtt{a}_{11}$ through $\mathtt{a}_{13}$), and unfolds to a sequential composition of procedure calls with $\mathtt{a}_{31}$ as argument.

The interesting observation is that none of the processes intervening in $\mathtt{elimRow}(\mathtt{A}_1,\mathtt{A}_3)$ occur in the expansion of $\mathtt{solve}(\mathtt{A'}_2)$. In other words,

`elimRow(A1,A3); solve(A'2)`

expands to

`elimOne(a12,a31,a32); elimOne(a13,a31,a33); elimOne(b1,a31,b3); set0(a31);`
`a21.c -> a22.div; a21.c -> a23.div; a21.c -> b2.div; a21 start x2; x2.1 -> a21.id`

and the semantics of PC allows the communications in the second line to be interleaved with those in the first line in any possible way. In the terminology of Cruz-Filipe and Montesi (2016a), the calls to `elimRow(A1,A3)` and `solve(A'2)` run in parallel.

This corresponds to implementing Gaussian elimination with pipelined communication and computation as in § 8.3 of (Grama et al., 2003). Indeed, as soon as any row has been reduced by all rows above it, it can apply `solve` to itself and try to begin reducing the rows below. It is a bit surprising that we get such parallel behaviour by straightforwardly implementing an imperative algorithm; the explanation is that EPP encapsulates the part of determining which communications can take place in parallel, removing this burden from the programmer.

6.3. Fast Fourier Transform

Our last example illustrates how we can compute the discrete Fourier transform of a vector in PC using the *Fast Fourier Transform* (FFT). We refer the reader to § 13.1 of (Grama et al., 2003) for details.

Definition 3. *Let $\vec{x} = \langle x_0, \ldots, x_{n-1} \rangle$ be a vector of n complex numbers. The* discrete Fourier transform *of $\vec{x}$ is $\vec{y} = \langle y_0, \ldots, y_{n-1} \rangle$, where $y_j = \sum_{k=0}^{n-1} x_k \omega^{kj}$ with $\omega = e^{2\pi i/n}$.*

Given $\vec{x}$, its discrete Fourier transform can be computed efficiently by the *Fast Fourier Transform* (FFT) as follows (Algorithm 13.1 in (Grama et al., 2003)). We assume n to be a power of 2; in the first call, ω has the value defined above.

procedure R_FFT(X,Y,n,ω)
if $n = 1$ then $y_0 = x_0$
  else R_FFT($\langle x_0, x_2, \ldots, x_{n-2} \rangle$,$\langle q_0, q_1, \ldots, q_{n/2} \rangle$,$n/2$,$\omega^2$)
    R_FFT($\langle x_1, x_3, \ldots, x_{n-1} \rangle$,$\langle t_0, t_1, \ldots, t_{n/2} \rangle$,$n/2$,$\omega^2$)
    for $j = 0$ to $n - 1$ do $y_j = q_{(j \% \frac{n}{2})} + \omega^j t_{(j \% \frac{n}{2})}$

To implement this procedure in PC, we need to communicate labels in selections to a group of processes as described in § 5. Our implementation thus uses the procedures `gselthen(p,Q)` and `gselelse(p,Q)`. The list functions we require (and which again can easily be seen to be acceptable list manipulating functions) are again `hd` and `tl`, together with `even` and `odd` (returning the elements on even or odd positions in the argument list, respectively) and `half1` and `half2` (returning the first or second half of the argument list, respectively).

We also use (without specifying them) the following auxiliary procedures.

- `intro(n,m,P)`, where `n` introduces `m` to every process in `P` (defined similarly to e.g. $\text{gsel}_{\text{then}}$)

- `power(n,m,nm)`, where at the end `nm` stores the result of exponentiating the value in `m` to the power of the value stored in `n` – see (Cruz-Filipe and Montesi, 2016a) for a possible implementation in a sublanguage of PC.

The one major difference between our implementation of FFT below and the algorithm R_FFT reported above is that we cannot create a variable number of fresh processes and pass them as arguments to other procedures (the auxiliary vectors $\vec{q}$ and $\vec{t}$). Instead, we use $\vec{y}$ to store the result of the recursive calls, and create two auxiliary processes inside each iteration of the final for loop.

```
fft(X,Y,n,w) = if n.is1
               then gsel_then(n,join(X,Y)); n -> w[then]; base(hd(X),hd(Y))
               else gsel_else(n,join(X,Y)); n -> w[else];
                    n start n'; n.half -> n'; intro(n,n',Y);
                    w start w'; w.square -> w'; intro(w,w',Y);
                    n: n' <-> w; w: n' <-> w';
                    fft(even(X),half1(Y),n',w');
                    fft(odd(X),half2(Y),n',w');
                    n' start wn; n': w <-> wn; power(n',w,wn);
                    w start wj; w.1 -> wj; intro(w,wj,Y);
                    combine(half1(Y),half2(Y),wn,w,wi)

base(x,y) = x.c -> y

combine(Y1,Y2,wn,w,wj) = combine_1(hd(Y1),hd(Y2),wn,wj);
                         w.c -> wj.mult;
                         combine(tl(Y1),tl(Y2),wn,w,wj)

combine_1(y1,y2,wn,wj) = y1 start q; y1.c -> q; y1: q <-> y2;
                         y2 start t; y2.c -> t; y2: t <-> y1; y2: t <-> wj;
                         q.c -> y1; wj.c -> t.mult; t.c -> y1.add;
                         q.c -> y2; wn.c -> t.mult; t.c -> y2.add
```

The level of parallelism in this implementation is suboptimal, as both recursive calls to `fft` use `n'` and `w'`. By duplicating these processes, these calls can run in parallel as in the previous example. (We chose the current formulation for simplicity.) Process `n'` is actually the main orchestrator of the whole execution.

7. Related Work

Choreographic Programming. Our examples cannot be written in previous models for choreographic programming, which lack full procedural abstraction. In state-of-the-art models such as those of Carbone et al. (2012) and Carbone and Montesi (2013), procedures cannot have continuations, there can only be a limited number of protocols running at any time (modulo dangling asynchronous actions), and the process names used in a procedure are statically determined. In PC, all these limitations are lifted.

Differently from PC, name mobility in choreographies is typically done using channel delegation, as shown by Carbone and Montesi (2013), which is less powerful: a process that introduces two other processes requires a new channel to communicate with them thenceforth.

Some choreography models include explicit parallel composition, $C \mid C'$. Most behaviours of $C \mid C'$ are already captured in PC; for example, $X\langle \mathsf{p}, \mathsf{q} \rangle \mid Y\langle \mathsf{r}, \mathsf{s} \rangle$ is equivalent to $X\langle \mathsf{p}, \mathsf{q} \rangle; Y\langle \mathsf{r}, \mathsf{s} \rangle$ in PC (cf. Example 3) – see (Carbone and Montesi, 2013) for a deeper discussion. If a parallel operator is desired, PC can be easily extended as in the work of Carbone et al. (2012).

In the work of Montesi and Yoshida (2013), choreographies can be integrated with existing process code (which can abstractly represent legacy code) by means of a type system, which we could easily integrate in PC.

Multiparty Session Types (MPST). In MPST, due to Honda et al. (2016), global types are choreographic specifications of single protocols, used for verifying the code of manually-written implementations in process models. Global types are similar to a simplified fragment of PC, obtained (among others) by replacing expressions and functions with constants (representing types), removing process creation (the processes are fixed), and restricting recursion to parameterless tail recursion.

MPST leaves protocol composition to the implementors of processes, which can result in deadlocks, unlike in PC. We illustrate this key difference using our syntax; we view a protocol in MPST as a (simplification of a) procedure in PC. Consider the protocols $X(\mathsf{r}, \mathsf{s}) = \mathsf{r}.e \texttt{ -> } \mathsf{s}.f$ and $Y(\mathsf{r}', \mathsf{s}') = \mathsf{r}'.e' \texttt{ -> } \mathsf{s}'.f'$, and their instantiations $X\langle \mathsf{p}, \mathsf{q} \rangle$ and $Y\langle \mathsf{q}, \mathsf{p} \rangle$. In MPST, protocols are projected separately and then they can be interleaved freely. Thus, a valid composition of the projections of these two protocols (in PP) is $\mathsf{p} \triangleright_v \mathsf{q}?f'; \mathsf{q}!e \mid \mathsf{q} \triangleright_v \mathsf{p}?f; \mathsf{p}!e'$. This network is obviously deadlocked, but MPST does not detect it because the interleaving of the two protocols is not checked. In PC, we can only obtain correct implementations, because compositions are defined at the level of choreographies, e.g., $X\langle \mathsf{p}, \mathsf{q} \rangle; Y\langle \mathsf{q}, \mathsf{p} \rangle$ or $Y\langle \mathsf{q}, \mathsf{p} \rangle; X\langle \mathsf{p}, \mathsf{q} \rangle$.

Deadlock-freedom for compositions in MPST can be obtained by restricting connections among processes participating in different protocols to form a tree, as shown by Carbone et al. (2016) and Carbone et al. (2017). This limits the communication structures that can be written, and is not necessary in PC, where there are no restrictions on the shape of connection graphs. Coppo et al. (2016) developed another technique for MPST using pre-orders, but this is also not as expressive as PC, as discussed by Cruz-Filipe and Montesi (2016b).

MPST can be extended to protocols where the number of participants is fixed only at runtime (Yoshida et al., 2010), or can grow during execution (Deniélou and Yoshida, 2011). These results use ad-hoc primitives and "middleware" terms in the process model, e.g., for tracking the number of participants in a session (Deniélou and Yoshida, 2011), which are not needed in PC. MPST can be nested (Demangeon and Honda, 2012), partially recalling our parametric procedures. Differently from PC, nested procedures in MPST are invoked by a coordinator (requiring extra communications), and compositions of such nested types can deadlock.

Asynchrony. It is also possible to endow PC with an asynchronous semantics that splits communication actions (value communications, label selections and

name-passing) into independent send and receive actions, which then may be interleaved with other actions through structural precongruence. The technique is a direct adaptation of the generic construction of Cruz-Filipe and Montesi (2017a): we introduce new runtime terms (not to be used by the programmer) to denote send and receive actions, extend structural precongruence to unfold communications into send/receive pairs, and prove that the resulting semantics does not essentially add new behaviours to the choreographies. Formally: each reduction step in the asynchronous semantics can always be completed to yield a reduction step in the original synchronous semantics. Likewise, the semantics of PP can be made asynchronous by adding a queue of incoming messages to each process and suitably changing the communication rules; again we can show that this semantics is a refinement of the synchronous semantics for PP.

Both progress (Theorem 1) and the EPP theorem (Theorem 4) readily extend to this new semantics. For progress, we simply observe that if reducing a typable choreography C yields a new choreography C' containing runtime terms, then C' can always reduce to another choreography C'' with no runtime terms, and C'' is also typable. For the EPP theorem, we need to extend EPP to the runtime terms – again following the strategy from (Cruz-Filipe and Montesi, 2017a) – and extend the proof of the theorem to the new cases. Both constructions are described in detail in the technical report (Cruz-Filipe and Montesi, 2016b), which also extends the type system of PC to the whole asynchronous language. As a consequence, choreography projections can never have orphan messages, in the sense that any message that is (asynchronously) sent can eventually be consumed by the intended receiver.

Sessions and Mobility. Recent theories based on session types, e.g., the works of Carbone et al. (2012), Carbone and Montesi (2013), Coppo et al. (2016), Honda et al. (2016) and Carbone et al. (2017), assume that all pairs of processes in a session have a private bidirectional channel to communicate. Thus, processes in a protocol must have a complete connection graph. PC can be used to reason about different kinds of network topologies.

Another important aspect of sessions is that each new protocol execution requires the creation of a new session, whereas procedure calls in PC reuse available connections – allowing for more efficient implementations. Our parallel downloader example uses this feature (Example 5).

The standard results of communication safety found in session-typed calculi can be derived from our EPP Theorem (Theorem 4), as discussed by Carbone and Montesi (2013).

Faults. We have abstracted from faults and divergence of internal computations: in PC, we assume that all internal computations terminate successfully. If we relax these conditions, deadlock-freedom can still be achieved simply by using timeouts and propagating faults through communications.

PC abstracts from the concrete locations of processes, but these may be useful in future studies on performance and fault recovery. Some interesting distributed bugs described by Leesatapornwongsa et al. (2016) are triggered by

unexpected fault conditions at nodes, making such faults an immediate candidate for the future developments of PC. Useful inspiration to this aim may be provided by (Capecchi et al., 2016).

8. Conclusions

We have presented Procedural Choreographies (PC), a language model for the development of correct-by-construction concurrent software based on message passing. PC advances the state of the art in choreographic programming towards realistic examples, by supporting modular programming through parametric procedures and by capturing networks with arbitrary connection structures (represented as graphs) and unbounded numbers of participants (processes). We conclude this article by describing some possible future directions.

An important future work will be to implement the new features explored in PC in current languages based on choreographic programming. The languages Chor (2017) and AIOCJ (Dalla Preda et al., 2014) are the respective implementations of the choreographic programming models due to Carbone and Montesi (2013) and Dalla Preda et al. (2017). Their compilers produce correct executable code in the Jolie language (Montesi et al., 2014) (for microservices). Both languages would benefit from full procedural abstraction, as investigated in this work.

Another future direction will be to explore how to apply PC to algorithms (or communication flows in general) that have to work in environments with restricted connection structures (e.g., due to hardware design). Specifically, we could use typing information on the connections used by a choreography to check whether such choreography can run on specific networks, e.g., hypercube or butterfly networks.

Finally, we wish to understand how to further extend the expressive power of PC. An important question is how we can express speculative parallelism, e.g., to program a process that contacts many other processes and then waits for the first available response from any of the contacted processes. Another interesting feature would be to support higher-order choreographies, by passing closures that describe the behaviour of multiple processes as parameters to procedures. This would allow us to compose choreographies in a functional style, rather than the imperative style that has been followed in choreographic programming so far.

Acknowledgements. This work was partly supported by the Danish Council for Independent Research, grant no. DFF–4005-00304, DFF-1323-00247 and DFF-7014-00041, and by the Open Data Framework project at the University of Southern Denmark.

References

Bravetti, M., Zavattaro, G., 2007. Towards a unifying theory for choreography conformance and contract compliance. In: Software Composition, 6th International Symposium, SC 2007, Braga, Portugal, March 24-25, 2007, Revised Selected Papers. pp. 34–50.
URL http://dx.doi.org/10.1007/978-3-540-77351-1_4

Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G., 2006. Choreography and orchestration conformance for system design. In: Proc. of Coordination. Vol. 4038 of LNCS. Springer, pp. 63–81.

Capecchi, S., Giachino, E., Yoshida, N., 2016. Global escape in multiparty sessions. Mathematical Structures in Computer Science 26 (2), 156–205.
URL http://dx.doi.org/10.1017/S0960129514000164

Carbone, M., Honda, K., Yoshida, N., 2012. Structured communication-centered programming for web services. ACM Trans. Program. Lang. Syst. 34 (2), 8.

Carbone, M., Lindley, S., Montesi, F., Schürmann, C., Wadler, P., 2016. Coherence generalises duality: A logical explanation of multiparty session types. In: CONCUR. Vol. 59 of LIPIcs. Schloss Dagstuhl, pp. 33:1–33:15.

Carbone, M., Montesi, F., 2013. Deadlock-freedom-by-design: multiparty asynchronous global programming. In: POPL. ACM, pp. 263–274.

Carbone, M., Montesi, F., Schürmann, C., 2018. Choreographies, logically. Distributed Computing 31 (1), 51–67.

Carbone, M., Montesi, F., Schürmann, C., Yoshida, N., 2017. Multiparty session types as coherence proofs. Acta Inf. 54 (3), 243–269.

Chor, 2017. Programming Language. http://www.chor-lang.org/.

Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L., 2016. Global progress for dynamically interleaved multiparty sessions. Mathematical Structures in Computer Science 26 (2), 238–302.
URL http://dx.doi.org/10.1017/S0960129514000188

Cruz-Filipe, L., Larsen, K. S., Montesi, F., 2017. The paths to choreography extraction. In: FoSSaCS. Vol. 10203 of Lecture Notes in Computer Science. pp. 424–440.

Cruz-Filipe, L., Montesi, F., 2016. Choreographies in practice. In: FORTE. Vol. 9688 of LNCS. Springer, pp. 114–123.

Cruz-Filipe, L., Montesi, F., 2016a. A core model for choreographic programming. In: FACS. Vol. 10231 of Lecture Notes in Computer Science. pp. 17–35.

Cruz-Filipe, L., Montesi, F., 2016b. A language for the declarative composition of concurrent protocols. CoRR abs/1602.03729.

Cruz-Filipe, L., Montesi, F., 2017a. On asynchrony and choreographies. In: Bartoletti, M., Bocchi, L., Henrio, L., Knight, S. (Eds.), Proceedings 10th Interaction and Concurrency Experience. Vol. 261 of EPTCS. Open Publishing Association, pp. 76–90.

Cruz-Filipe, L., Montesi, F., 2017b. Procedural choreographic programming. In: Bouajjani, A., Silva, A. (Eds.), FORTE. Vol. 10321 of LNCS. Springer, pp. 92–107.

Dalla Preda, M., Gabbrielli, M., Giallorenzo, S., Lanese, I., Mauro, J., 2017. Dynamic choreographies: Theory and implementation. Logical Methods in Computer Science 13 (2).

Dalla Preda, M., Giallorenzo, S., Lanese, I., Mauro, J., Gabbrielli, M., 2014. AIOCJ: A choreographic framework for safe adaptive distributed applications. In: Combemale, B., Pearce, D., Barais, O., Vinju, J. (Eds.), SLE. Vol. 8706 of LNCS. Springer, pp. 161–170.

Demangeon, R., Honda, K., 2012. Nested protocols in session types. In: CONCUR. pp. 272–286.

Deniélou, P.-M., Yoshida, N., 2011. Dynamic multirole session types. In: POPL. ACM, pp. 435–446.

Grama, A., Gupta, A., Karypis, G., Kumar, V., 2003. Introduction to Parallel Computing. Pearson, 2nd edition.

Honda, K., Mukhamedov, A., Brown, G., Chen, T.-C., Yoshida, N., 2011. Scribbling interactions with a formal foundation. In: ICDCIT. Springer, pp. 55–75.

Honda, K., Vasconcelos, V., Kubo, M., 1998. Language primitives and type disciplines for structured communication-based programming. In: ESOP. Vol. 1381 of LNCS. Springer, pp. 122–138.

Honda, K., Yoshida, N., Carbone, M., 2016. Multiparty Asynchronous Session Types. J. ACM 63 (1), 9.
URL `http://doi.acm.org/10.1145/2827695`

International Telecommunication Union, 1996. Recommendation Z.120: Message sequence chart.

JBoss Community and Red Hat, 2017. SAVARA and Testable Architecture. `http://www.jboss.org/savara/`.

Lanese, I., Guidi, C., Montesi, F., Zavattaro, G., 2008. Bridging the gap between interaction- and process-oriented choreographies. In: SEFM. pp. 323–332.

Lanese, I., Montesi, F., Zavattaro, G., 2013. Amending choreographies. In: WWV. pp. 34–48.

Lange, J., Tuosto, E., Yoshida, N., 2015. From communicating machines to graphical choreographies. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 221–232.

Leesatapornwongsa, T., Lukman, J. F., Lu, S., Gunawi, H. S., 2016. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In: ASPLOS. ACM, pp. 517–530.

Lluch-Lafuente, A., Nielson, F., Nielson, H. R., 2015. Discretionary information flow control for interaction-oriented specifications. In: Logic, Rewriting, and Concurrency. Vol. 9200 of Lecture Notes in Computer Science. Springer, pp. 427–450.

López, H. A., Heussen, K., 2017. Choreographing cyber-physical distributed control systems for the energy sector. In: SAC. ACM, pp. 437–443.

López, H. A., Nielson, F., Nielson, H. R., 2016. Enforcing availability in failure-aware communicating systems. In: FORTE. Vol. 9688 of Lecture Notes in Computer Science. Springer, pp. 195–211.

Montesi, F., 2013. Choreographic Programming. Ph.D. Thesis, IT University of Copenhagen, http://fabriziomontesi.com/files/choreographic_programming.pdf .

Montesi, F., Guidi, C., Zavattaro, G., 2014. Service-oriented programming with jolie. In: Web Services Foundations. pp. 81–107.
URL `http://dx.doi.org/10.1007/978-1-4614-7518-7_4`

Montesi, F., Yoshida, N., 2013. Compositional choreographies. In: CONCUR. Vol. 8052 of LNCS. Springer, pp. 425–439.

MPI Forum, 2015. MPI: A Message-Passing Interface Standard. High-Performance Computing Center Stuttgart, version 3.1.

Needham, R., Schroeder, M., Dec. 1978. Using encryption for authentication in large networks of computers. Commun. ACM 21 (12), 993–999.

Object Management Group, 2017. Business Process Model and Notation. `http://www.omg.org/spec/BPMN/2.0/`.

Qiu, Z., Zhao, X., Cai, C., Yang, H., 2007. Towards the theoretical foundation of choreography. In: WWW. ACM, pp. 973–982.

Sangiorgi, D., Walker, D., 2001. The π-calculus: a Theory of Mobile Processes. Cambridge University Press.

W3C WS-CDL Working Group, 2004. Web services choreography description language version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/.

Yoshida, N., Deniélou, P.-M., Bejleri, A., Hu, R., 2010. Parameterised multiparty session types. In: FOSSACS. Vol. 6014 of LNCS. Springer, pp. 128–145.

Appendix A. Detailed definitions and proofs

We report on proofs of results that were omitted from the main part of this paper.

Appendix A.1. Type checking and type inference

To prove this theorem, we begin by establishing some technical lemmas about typing.

Lemma 1 (Monotonicity)**.** *Let Γ and Γ' be typing contexts with $\Gamma \subseteq \Gamma'$, G_1, G_1' and G be connection graphs such that $G_1 \subseteq G$, and C be a choreography. If $\Gamma; G_1 \vdash C \triangleright G_1'$, then $\Gamma'; G \vdash C \triangleright G \cup G_1'$.*

Proof. Straightforward by induction on the derivation of $\Gamma; G_1 \vdash C \triangleright G_1'$. □

Lemma 2 (Sequentiality)**.** *Let Γ be a typing context, G_1, G_1', G_2 and G_2' be connection graphs such that $G_2 \subseteq G_1'$, and C_1, C_2 be choreographies. If $\Gamma; G_1 \vdash C_1 \triangleright G_1'$ and $\Gamma; G_2 \vdash C_2 \triangleright G_2'$, then $\Gamma; G_1 \vdash C_1 \fatsemi C_2 \triangleright G_1' \cup G_2'$.*

Proof. Straightforward by induction on the derivation of $\Gamma; G_2 \vdash C_2 \triangleright G_2'$. □

Lemma 3 (Substitution)**.** *Let Γ be a typing context, G and G' be connection graphs, and C be a choreography. Let $\tilde{\mathsf{p}}$ be a set of process names that are free in C and $\tilde{\mathsf{q}}$ be a set of process names that do not occur (free or bound) in C. If $\Gamma; G \vdash C \triangleright G'$, then $\Gamma[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}]; G[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \vdash C[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \triangleright G'[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}]$.*

Proof. Straightforward by induction on the derivation of $\Gamma; G \vdash C \triangleright G'$, as all typing rules are valid when substitutions are applied. □

We are now ready to start proving Theorem 1. The following lemma takes care of the base cases, and is required for one of the inductive steps.

Lemma 4. *Let Γ be a set of typing judgements, $\mathscr{D}$ a set of procedure definitions, G_1 and G_1' connection graphs, and C a choreography that does not start with $\mathbf{0}$ or a procedure call. Assume that $\Gamma \vdash \mathscr{D}$ and $\Gamma; G_1 \vdash C \triangleright G_1'$. For every state σ, there exist Γ', σ', C', G_2 and G_2' such that $G_1, C, \sigma \to_{\mathscr{D}} G_2, C', \sigma'$ and $\Gamma'; G_2 \vdash C' \triangleright G_2'$.*

Proof. By case analysis on the last step of the proof of $\Gamma; G_1 \vdash C \triangleright G_1'$. By hypothesis, this proof cannot end with an application of rules $\lfloor$T|End$\rceil$, $\lfloor$T|EndSeq$\rceil$ or $\lfloor$T|Call$\rceil$; we detail all cases for completeness, but the only non-trivial one is the last.

- $\lfloor$T|Start$\rceil$: then C is $\mathsf{p}\,\mathsf{start}\,\mathsf{q}^T; C^\circ$ and by hypothesis

$$\Gamma, q : T; G_1 \cup \{p \leftrightarrow q\} \vdash C^\circ \triangleright G_1' \,.$$

  Since $G_1, \mathsf{p}\,\mathsf{start}\,\mathsf{q}^T; C, \sigma \to_{\mathscr{D}} G_1 \cup \{p \leftrightarrow q\}, C^\circ, \sigma[\mathsf{q} \mapsto \bot_T]$ by rule $\lfloor$C|Start$\rceil$, taking $\Gamma' = \Gamma, \mathsf{q} : T$, $\sigma' = \sigma[\mathsf{q} \mapsto \bot_T]$, $C' = C^\circ$, $G_2 = G_1 \cup \{p \leftrightarrow q\}$ and $G_2' = G_1'$ establishes the thesis.

- $\lfloor$T|Com$\rceil$: then C is $\mathsf{p}.e$ -> $\mathsf{q}.f; C^\circ$ and by hypothesis $p \xleftrightarrow{G_1} q$, $f[\sigma(\mathsf{q})/*](e[\sigma(\mathsf{p})/*])$ is a valid expression of type T_q, and $\Gamma; G_1 \vdash C \triangleright G'_1$. Then all the preconditions of $\lfloor$C|Com$\rceil$ are met, so taking $\Gamma' = \Gamma$, $\sigma' = \sigma[\mathsf{q} \mapsto w]$ where $e \downarrow^{\mathsf{p}}_{\sigma} v$ and $f(v) \downarrow^{\mathsf{q}}_{\sigma} w$, $C' = C^\circ$, $G_2 = G_1$ and $G'_2 = G'_1$ establishes the thesis.

- $\lfloor$T|Sel$\rceil$: then C is p -> $\mathsf{q}[\ell]; C^\circ$ and by hypothesis $p \xleftrightarrow{G_1} q$ and $\Gamma; G_1 \vdash C \triangleright G'_1$. By $\lfloor$C|Sel$\rceil$, G_1, p -> $\mathsf{q}[\ell]; C^\circ, \sigma \rightarrow_{\mathscr{D}} G_1, C^\circ, \sigma$, so taking $\Gamma' = \Gamma$, $\sigma' = \sigma$, $C' = C^\circ$, $G_2 = G_1$ and $G'_2 = G'_1$ again establishes the thesis.

- $\lfloor$T|Tell$\rceil$: then C is p: q <-> $\mathsf{r}; C^\circ$ and by hypothesis both $p \xleftrightarrow{G_1} q$, $p \xleftrightarrow{G_1} r$, and $\Gamma; G_1 \cup \{q \leftrightarrow r\} \vdash C^\circ \triangleright G'_1$. Since the preconditions of rule $\lfloor$C|Tell$\rceil$ are met, by taking $\Gamma' = \Gamma$, $\sigma' = \sigma$, $C' = C^\circ$, $G_2 = G_1 \cup \{q \leftrightarrow r\}$ and $G'_2 = G'_1$ establishes the thesis.

- $\lfloor$T|Cond$\rceil$: then C is if $\mathsf{p}.e$ then C_1 else $C_2; C^\circ$ and by hypothesis $e[\sigma(\mathsf{p})/*]$ is a valid Boolean expression, $\Gamma; G_1 \vdash C_i \triangleright G^\circ_i$ and $\Gamma; G^\circ_1 \cap G^\circ_2 \vdash C^\circ \triangleright G'_1$.

  Suppose $e \downarrow^{\mathsf{p}}_{\sigma}$ true (the other case is similar). Then G_1, if $\mathsf{p}.e$ then C_1 else $C_2; C^\circ, \sigma \rightarrow_{\mathscr{D}} G_1, C_1 \fatsemi C, \sigma$. Since $G^\circ_1 \cap G^\circ_2 \subseteq G^\circ_1$, Lemma 2 allows us to conclude that $\Gamma; G_1 \vdash C_1 \fatsemi C \triangleright G'_1 \cup G^\circ_1$, whence the thesis follows by taking $\Gamma' = \Gamma$, $C' = C_1 \fatsemi C$, $G_2 = G_1$ and $G'_2 = G'_1 \cup G^\circ_1$.

□

Proof (Theorem 1). If $C \preceq_{\mathscr{D}} \mathbf{0}$, then the first case holds. Assume that $C \not\preceq_{\mathscr{D}} \mathbf{0}$; we show that the second case holds by induction on the proof of $\Gamma; G_1 \vdash C \triangleright G'_1$. By hypothesis, the last rule applied in this proof cannot be $\lfloor$T|End$\rceil$; the cases where the last rule applied is $\lfloor$T|Start$\rceil$, $\lfloor$T|Com$\rceil$, $\lfloor$T|Sel$\rceil$, $\lfloor$T|Tell$\rceil$ or $\lfloor$T|Cond$\rceil$ follow immediately from Lemma 4, while the case of rule $\lfloor$T|EndSeq$\rceil$ is straightforward from the induction hypothesis.

We focus on the case of rule $\lfloor$T|Call$\rceil$. In this case, C has the form $X\langle\tilde{\mathsf{p}}\rangle; C^\circ$, and we know that $\Gamma \vdash X(\widetilde{\mathsf{q}^T}) : (G_X \triangleright G'_X)$, $\Gamma \vdash \tilde{\mathsf{p}} : T$, $G_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \subseteq G_1$ and $\Gamma; G_1 \cup (G'_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}]) \vdash C^\circ \triangleright G'_1$. Let C_X be the body of X as defined in $\mathscr{D}$; from the hypothesis that $\Gamma \vdash \mathscr{D}$ we also know that $\Gamma_X; G_X \vdash C_X \triangleright G''_X$ for some G''_X such that G'_X is G''_X restricted to the set $\tilde{\mathsf{p}}$. By Lemma 3, $\Gamma_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}]; G_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \vdash C_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \triangleright G''_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}]$, whence by Lemma 1 also $\Gamma; G_1 \vdash C_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \triangleright G''_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \cup G_1$. By applying rule $\lfloor$C|Unfold$\rceil$, we conclude that $X\langle\tilde{\mathsf{p}}\rangle; C^\circ \preceq_{\mathscr{D}} C_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \fatsemi C^\circ$, and Lemma 2 allows us to conclude that $\Gamma; G_1 \vdash C_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \fatsemi C^\circ \triangleright G'_1$. Since procedure calls are guarded, C_X does not begin with a procedure call, and Lemma 4 establishes the thesis. □

We now present an alternative proof of Theorem 2 that constructs the graphs G_X and G'_X required for typing each procedure body using the information from the choreography to be typed.

Theorem 2. The key step in this proof is showing, given Γ and $\langle\mathscr{D}, C\rangle$, how to find a "canonical typing" for the recursive definitions, the set $\Gamma_{\mathscr{D}}$, such that $\Gamma_{\mathscr{D}} \vdash \mathscr{D}$ and $\Gamma, \Gamma_{\mathscr{D}}; G_C \vdash C \triangleright G'$ (with G' inferred) iff $\Gamma, \Gamma'; G_C \vdash C \triangleright G''$ for

some Γ' and G''. More precisely, we need to find graphs G_X and G'_X for each procedure X defined in $\mathscr{D}$.

Our proof proceeds in three steps. First, for each X we compute an underapproximation G°_X of the output graph G'_X, containing all the relevant connections that executing X can add. Using this, we compute the input graph G_X and the output graph $G'_X = G_X \cup G^\circ_X$. Both these steps are achieved by computing a minimal fixpoint of a monotonic operator in the set of all graphs whose vertices are the parameters of X. Finally, we argue that the typing $X : G_X \triangleright G_X$ is minimal, and therefore the set $\Gamma_{\mathscr{D}}$ of all such typings fulfills the property we require.

Throughout the remainder of this proof, we assume $\mathscr{D} = \{X_i(\widetilde{\mathsf{q}^i}) = C_i \mid i = 1, \ldots, n\}$.

1. In order to compute $G^\circ_{X_i}$, we define an auxiliary function fwd with intended meaning as follows: $\mathsf{fwd}^{\tilde{G}_i}_{C_j}(G)$ computes the communication graph obtained from G after one execution of the body of X_j, assuming that $X_i(\widetilde{\mathsf{q}_i}) : \emptyset \triangleright G_i$ for all i and ignoring newly created processes. We use a conditional union operator $\uplus$ where $G \uplus \{e\}$ denotes $G \cup \{e\}$ if e is an edge connecting two vertices in G, and G otherwise. The function fwd is defined as follows.

$$\mathsf{fwd}^{\tilde{G}_i}_{\mathbf{0}}(G) = G$$
$$\mathsf{fwd}^{\tilde{G}_i}_{\mathbf{0};C}(G) = \mathsf{fwd}^{\tilde{G}_i}_{C}(G)$$
$$\mathsf{fwd}^{\tilde{G}_i}_{\mathsf{p}.e\ \texttt{->}\ \mathsf{q}.f;C}(G) = \mathsf{fwd}^{\tilde{G}_i}_{C}(G)$$

$$\mathsf{fwd}^{\tilde{G}_i}_{\mathsf{p}\ \texttt{->}\ \mathsf{q}[\ell];C}(G) = \mathsf{fwd}^{\tilde{G}_i}_{C}(G)$$
$$\mathsf{fwd}^{\tilde{G}_i}_{\mathsf{p}\ \mathsf{start}\ \mathsf{q}^T;C}(G) = \mathsf{fwd}^{\tilde{G}_i}_{C}(G)$$
$$\mathsf{fwd}^{\tilde{G}_i}_{\mathsf{p}:\mathsf{q}\ \texttt{<->}\ \mathsf{r};C}(G) = \mathsf{fwd}^{\tilde{G}_i}_{C}(G \uplus \{\mathsf{q} \leftrightarrow \mathsf{r}\})$$
$$\mathsf{fwd}^{\tilde{G}_i}_{\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2;C}(G) = \mathsf{fwd}^{\tilde{G}_i}_{C}(\mathsf{fwd}^{\tilde{G}_i}_{C_1}(G) \cap \mathsf{fwd}^{\tilde{G}_i}_{C_2}(G))$$
$$\mathsf{fwd}^{\tilde{G}_i}_{X_i\langle\tilde{\mathsf{p}}\rangle;C}(G) = \mathsf{fwd}^{\tilde{G}_i}_{C}(G \uplus G_i[\tilde{\mathsf{p}}/\widetilde{\mathsf{q}^i}])$$

Using fwd, we define an operator $\mathscr{T}_{\mathsf{fwd}}$ over the set $\mathscr{G}$ of tuples of graphs over the parameters of X_i, i.e. $\mathscr{G} = \{\widetilde{G_i} \mid G_i \text{ is a graph over } \widetilde{\mathsf{q}^i}\}$. Observe that $\mathscr{G}$ is a complete lattice wrt componentwise inclusion.

$$\mathscr{T}_{\mathsf{fwd}}(\widetilde{G_i}) = \widetilde{\mathsf{fwd}^{\tilde{G}_i}_{C_i}(G_i)}$$

This operator is monotonic, since fwd only adds edges to its argument, and thus has a least fixpoint that can be computed by iterating $\mathscr{T}_{\mathsf{fwd}}$ from the tuple of empty graphs over the right sets of vertices. Furthermore, since $\mathscr{G}$ is finite (each graph has a finite number of vertices) this fixpoint corresponds to a finite iterate, and can thus be computed in finite time. We denote this fixpoint by $\widetilde{G^\circ_{X_i}}$.

$$
\begin{aligned}
\mathsf{bck}^{\tilde{G}_i}_{\mathbf{0}}(G_a, G_b) &= \langle G_a, G_b \rangle \\
\mathsf{bck}^{\tilde{G}_i}_{\mathbf{0};C}(G_a, G_b) &= \mathsf{bck}^{\tilde{G}_i}_{C}(G_a, G_b) \\
\mathsf{bck}^{\tilde{G}_i}_{\mathsf{p}.e \;\texttt{->}\; \mathsf{q}.f;C}(G_a, G_b) &= \begin{cases} \mathsf{bck}^{\tilde{G}_i}_{C}(G_a, G_b) & \text{if } \mathsf{p} \leftrightarrow \mathsf{q} \in G_b \\ \mathsf{bck}^{\tilde{G}_i}_{C}(G_a \uplus \{\mathsf{p} \leftrightarrow \mathsf{q}\}, G_b \uplus \{\mathsf{p} \leftrightarrow \mathsf{q}\}) & \text{otherwise} \end{cases} \\
\mathsf{bck}^{\tilde{G}_i}_{\mathsf{p} \;\texttt{->}\; \mathsf{q}[\ell];C}(G_a, G_b) &= \begin{cases} \mathsf{bck}^{\tilde{G}_i}_{C}(G_a, G_b) & \text{if } \mathsf{p} \leftrightarrow \mathsf{q} \in G_b \\ \mathsf{bck}^{\tilde{G}_i}_{C}(G_a \uplus \{\mathsf{p} \leftrightarrow \mathsf{q}\}, G_b \uplus \{\mathsf{p} \leftrightarrow \mathsf{q}\}) & \text{otherwise} \end{cases} \\
\mathsf{bck}^{\tilde{G}_i}_{\mathsf{p}\;\mathsf{start}\;\mathsf{q}\,T\,;C}(G_a, G_b) &= \mathsf{bck}^{\tilde{G}_i}_{C}(G_a, G_b) \\
\mathsf{bck}^{\tilde{G}_i}_{\mathsf{p}:\mathsf{q} \;\texttt{<->}\; \mathsf{r};C}(G_a, G_b) &= \begin{cases} \mathsf{bck}^{\tilde{G}_i}_{C}(G_a, G_b \uplus \{\mathsf{q} \leftrightarrow \mathsf{r}\}) & \text{if } \mathsf{p} \leftrightarrow \mathsf{q}, \mathsf{p} \leftrightarrow \mathsf{r} \in G_b \\ \mathsf{bck}^{\tilde{G}_i}_{C}(G_a \uplus \{\mathsf{p} \leftrightarrow \mathsf{q}\}, G_b \uplus \{\mathsf{p} \leftrightarrow \mathsf{q}, \mathsf{q} \leftrightarrow \mathsf{r}\}) & \text{if } \mathsf{p} \leftrightarrow \mathsf{q} \notin G_b,\ \mathsf{p} \leftrightarrow \mathsf{r} \in G_b \\ \mathsf{bck}^{\tilde{G}_i}_{C}(G_a \uplus \{\mathsf{p} \leftrightarrow \mathsf{r}\}, G_b \uplus \{\mathsf{p} \leftrightarrow \mathsf{r}, \mathsf{q} \leftrightarrow \mathsf{r}\}) & \text{if } \mathsf{p} \leftrightarrow \mathsf{q} \in G_b,\ \mathsf{p} \leftrightarrow \mathsf{r} \notin G_b \\ \mathsf{bck}^{\tilde{G}_i}_{C}(G_a \uplus \{\mathsf{p} \leftrightarrow \mathsf{q}, \mathsf{p} \leftrightarrow \mathsf{r}\}, G_b \uplus \{\mathsf{p} \leftrightarrow \mathsf{q}, \mathsf{p} \leftrightarrow \mathsf{r}, \mathsf{q} \leftrightarrow \mathsf{r}\}) & \text{if } \mathsf{p} \leftrightarrow \mathsf{q}, \mathsf{p} \leftrightarrow \mathsf{r} \notin G_b \end{cases} \\
\mathsf{bck}^{\tilde{G}_i}_{\mathsf{if}\;\mathsf{p}.e\;\mathsf{then}\;C_1\;\mathsf{else}\;C_2;C}(G_a, G_b) &= \mathsf{bck}^{\tilde{G}_i}_{C}(\mathsf{fst}(\mathsf{bck}^{\tilde{G}_i}_{C_1}(G_a, G_b)) \cup \mathsf{fst}(\mathsf{bck}^{\tilde{G}_i}_{C_2}(G_a, G_b)), \mathsf{snd}(\mathsf{bck}^{\tilde{G}_i}_{C_1}(G_a, G_b)) \cap \mathsf{snd}(\mathsf{bck}^{\tilde{G}_i}_{C_2}(G_a, G_b))) \\
\mathsf{bck}^{\tilde{G}_i}_{X_i\langle \tilde{\mathsf{p}} \rangle;C}(G_a, G_b) &= \mathsf{bck}^{\tilde{G}_i}_{C}(G_a \uplus (G_i[\tilde{\mathsf{p}}/\widetilde{\mathsf{q}^i}] \setminus G_b), G_b \uplus G_i[\tilde{\mathsf{p}}/\widetilde{\mathsf{q}^i}] \uplus G_i^\circ[\tilde{\mathsf{p}}/\widetilde{\mathsf{q}^i}])
\end{aligned}
$$

Figure A.10: Definition of bck (case 2 in the proof of Theorem 2).

2. The construction of the input graphs G_{X_i} follows the same idea: we go through the C_is noting the edges that are required for all communications to be able to take place. It is however slightly more complicated, because we have to keep track of edges that the choreography adds to the graph; we therefore need a function bck that manipulates two graphs instead of one. More precisely, $\mathsf{bck}^{\tilde{G}_i}_{C}(G, G)$ returns the graph extending G that is needed for correctly executing C (ignoring newly created processes); the first argument keeps track of the edges that need to be added to G, and the second argument keeps track of edges added by executing C. This function uses the graphs $G^\circ_{X_i}$ computed earlier, which explains why it has to be defined afterwards. We use the same notational conventions as above, and let $\mathsf{fst}\langle a, b\rangle = a$ and $\mathsf{snd}\langle a, b\rangle = b$. The definition of bck is given in Figure A.10; it is not the simplest possible, but the formulation given is sufficient for our purposes.

   Again we define a monotonic operator over the same $\mathscr{G}$ as above.

$$\mathscr{T}_{\mathsf{bck}}(\widetilde{G_i}) = \mathsf{fst}(\widetilde{\mathsf{bck}^{\tilde{G}_i}_{C_i}(G_i, G_i)})$$

   We do not need to recompute G°_i, since these graphs contain all edges that can possibly be added by executing C_i. The least fixpoint of $\mathscr{T}_{\mathsf{bck}}$ can again be computed by finitely iterating this operator, and it is precisely $\widetilde{G_{X_i}}$. We then define $G'_{X_i} = G_{X_i} \cup G^\circ_{X_i}$.

3. We now show that $\Gamma_{\mathscr{D}} = \{X_i(\widetilde{\mathsf{q}_i}) : G_{X_i} \triangleright G'_{X_i}\}$ is a minimal typing of $\mathscr{D}$, in the sense explained earlier. Observe that it is possible that $\Gamma_{\mathscr{D}} \not\vdash \mathscr{D}$, in particular if the X_i are ill-formed choreographies.

Suppose that $\Gamma, \Gamma'; G_C \vdash C \triangleright G$ for some Γ' and G. We argue that $\Gamma, \Gamma_{\mathscr{D}}; G_C \vdash C \triangleright G'$, where G' is inferred from the typing rules. For each procedure $X_i(\widetilde{\mathsf{q}_i}) = C_i$, there must be a unique typing $X_i(\widetilde{\mathsf{q}_i}) : G^*_{X_i} \triangleright G^{**}_{X_i}$ in Γ'. By a simple inductive argument one can show that $G^\circ_{X_i} \subseteq G^{**}$ (since $\emptyset \subseteq G^{**}_{X_i}$ and $\mathscr{T}_{\mathsf{fwd}}$ preserves inclusion in $G^{**}_{X_i}$). Similarly, one shows that $G_{X_i} \subseteq G^*_{X_i}$ and that $G^{**}_{X_i} \setminus G^*_{X_i} \subseteq G'_{X_i} \setminus G_{X_i}$. As a consequence, the typing derivation for $\Gamma, \Gamma'; G_C \vdash C \triangleright G$ can be used for $\Gamma, \Gamma_{\mathscr{D}}; G_C \vdash C \triangleright G'$, as all applications of rule $\lfloor$T|Call$\rceil$ are guaranteed to be valid (their preconditions hold) and to produce the same results (they change the communication graph in the same way). $\square$