Formalising a Turing-Complete Choreographic Language in Coq

Luís Cruz-Filipe
Dept. Mathematics and Computer Science, Univ. Southern Denmark, Odense, Denmark

Fabrizio Montesi
Dept. Mathematics and Computer Science, Univ. Southern Denmark, Odense, Denmark

Marco Peressotti
Dept. Mathematics and Computer Science, Univ. Southern Denmark, Odense, Denmark

Abstract

The theory of choreographic languages typically includes a number of complex results that are proved by structural induction. The high number of cases and the subtle details in some of them lead to long reviewing processes, and occasionally to errors being found in published proofs. In this work, we take a published proof of Turing completeness of a choreographic language and formalise it in Coq. Our development includes formalising the choreographic language, its basic properties, Kleene's theory of partial recursive functions, the encoding of these functions as choreographies, and a proof that this encoding is correct.

With this effort, we show that theorem proving can be a very useful tool in the field of choreographic languages: besides the added degree of confidence that we get from a mechanised proof, the formalisation process led us to a significant simplification of the underlying theory. Our results offer a foundation for the future formal development of choreographic languages.

2012 ACM Subject Classification Theory of computation → Process calculi; Theory of computation → Recursive functions; Theory of computation → Logic and verification

Keywords and phrases Choreographic Programming, Formalisation, Turing Completeness

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.11

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.4479102`

Funding Work partially supported by Villum Fonden, grant no. 29518.

1 Introduction

Background. In the setting of concurrent and distributed systems, choreographic languages are used to define interaction protocols that communicating processes should abide to [21, 31, 34]. These languages are akin to the "Alice and Bob" notation found in security protocols, and inherit the key idea of making movement of data manifest in programs [30]. This is usually obtained through a linguistic primitive like `Alice.e` → `Bob.x`, read "`Alice` communicates the result of evaluating expression `e` to `Bob`, which stores it in its local variable `x`".

In recent years, the communities of concurrency theory and programming languages have been prolific in developing methodologies based on choreographies, yielding results in program verification, monitoring, and program synthesis [2, 20]. For example, in *multiparty session types*, types are choreographies used for checking statically that a system of processes implements protocols correctly [19]. Further, in *choreographic programming*, choreographic languages are elevated to full-fledged programming languages [28], which can express how data should be pre- and post-processed by processes (encryption, validation, anonymisation, etc.). Choreographic programming languages showed promise in a number of contexts, including parallel algorithms [7], cyber-physical systems [26, 25, 17], self-adaptive systems [12], system integration [16], information flow [23], and the implementation of security protocols [17].

The problem. Proofs in the field of choreographic languages are extremely technical. They have to cover many cases, and they typically involve translations from/to other languages that come with their own structures and semantics. The level of complexity makes peer-reviewing challenging. For example, it has recently been discovered that a significant number (at least 5) of key results published in peer-reviewed articles on multiparty session types actually do not hold, and that their statements require modification [32].

This article. The aim of this article is to show that computer-aided verification – in particular, interactive theorem proving – can be successfully applied to the study of choreographies and to provide solid foundations for future developments. Before presenting our scientific contributions, it is interesting to look at the story behind this article, as it tells us that interactive theorem proving is not just a tool to check what we already know.

Our development started in late 2018. Its starting point was the theory of Core Choreographies (CC), a minimalistic language that we previously proposed for the study of choreographic programming [8]. CC is designed to include only the essential features of choreographic languages and minimal computational capabilities at processes (computing the successor of a natural number and deciding equality of two natural numbers). Nevertheless, it is expressive enough to be Turing complete, which is proven by developing a provably-correct translation of Kleene's partial recursive functions [22] into choreographies that implement the source functions by means of communication [8].

At the TYPES conference in 2019, we gave an informal progress report on the formalisation of CC using the Coq theorem prover [9]. Our effort revealed soon a crux of unparsimonious complexity in the theory: a set of term-rewriting rules for a precongruence relation used in the semantics of the language for (i) expanding procedure calls with the bodies of their respective procedures and (ii) reshuffling independent communications in choreographies to represent correctly concurrent execution. This relation is closed under context and transitivity, and it can always be involved in the derivation of reductions, which led to tedious induction on the derivation of these term rewritings in almost all cases of proofs that had to do with the semantics of choreographies. In addition to being time consuming, formalising this aspect makes the theory presented in [8] much more complicated (we discuss this in Section 2.4).

At the time, the second author had been teaching a few editions of a course that includes theory of choreographies. Interestingly, the same technical aspects that made the formalisation of CC much more intricate than its original theory were all found to be subtly complicated by students. Motivated by this observation and our early efforts in formalising [8], this author developed a revisited theory of CC for his course material that dispenses with the problematic notions and shows that they are actually *unnecessary* [29]. The choreography theory in [29] is the one that we deal with in this article.

Thus, besides its scientific contributions, this article also shows that theorem proving can be used in research: the insights that we got doing this formalisation led to changes in the original theory. We show that this did not come at the cost of expressive power: the translation of Kleene's partial recursive functions from [8] still works as-is for the theory in [29]. Furthermore, while formalising the theory we realised that some assumptions in some results were actually not necessary, yielding stronger results.

Contributions. This article presents the first formalised theory of a full-fledged choreographic language, including its syntax and semantics, and the main properties of determinism, confluence, and deadlock-freedom by design. This theory is formalised in Coq, using its module system to make it parametric. We show that the choreographic language is Turing

complete, by encoding Kleene's partial recursive functions as choreographies and proving this encoding sound. The full development can be downloaded at [11].

Structure. For compactness we state our definitions and results in Coq syntax. In Section 2, we present the syntax and semantics of our choreographic language, based on its formalisation in Coq, and establish the main theoretical properties of this language. Section 3 presents the theory and formalisation of Kleene's partial recursive functions, and Section 4 describes their encoding as choreographies and the proof of Turing completeness of the choreographic language. We discuss the relevance of our results and future work in Section 5.

2 Choreographies

In this section, we introduce the choreographic language of Core Choreographies (CC), together with the corresponding formalisation.

A choreography specifies a protocol involving different participants (processes) that can communicate among themselves and possess local computational capabilities. Each process also has storage, which is accessible through variables. There are two kinds of communications: value communications, where the sender process locally evaluates an expression and sends the result to the receiver process, who stores it in one of its variables; and label selection, where the sender selects one among some alternative behaviours (tagged by labels) offered by the receiver. A choreography can also define (recursive) procedures, which can be invoked by their name anywhere. The formal syntax of choreographies is given in Section 2.2.

2.1 Preliminaries

We define the type of choreographies as a parametric Coq `Module`, taking eight parameters: the types of process identifiers (processes for short) `Pid`, local variables `Var` (used to access the processes' storage), values `Val`, expressions `Expr`, Boolean expressions `BExpr`, procedure names `RecVar` (from *recursion variables*), and the evaluation functions mapping expressions to values and Boolean expressions to Booleans.

The first six parameters are datatypes that are equipped with a decidable equality. Due to difficulties with using the definitions in Coq's standard libraries, we reimplemented this as a `Module Type DecType`, and defined a functor `DecidableType` providing the usual lemmas to simplify function definitions by case analysis on equality of two objects.

Evaluation requires a *local state*, mapping process variables to actual values. We model states as functors, taking `Var` and `Val` as parameters and returning a `Module` containing this function type together with an operator to update the state (by changing the value assigned to one variable) and lemmas characterising it. For the semantics of choreographies, we also need *global states*, which take `Pid` as an additional parameter and map each process to a local state. This type is again enriched with operations to update a global state.

Both modules include a definition of extensional equality: for two local states `s` and `s'`, `eq_state s s'` holds iff $\forall$ `x`, `s x` $=$ `s' x`, and for global states `eq_state_ext s s'` holds iff $\forall$ `p`, `eq_state (s p) (s' p)`.

An evaluation function is a function mapping expressions to values, given a local state. Evaluation must be compatible with extensional equality on states.

```
Module Type Eval (Expression Vars Input Output : DecType).

Parameter eval : Expression.t → (Vars.t → Input.t) → Output.t.
Parameter eval_wd : ∀ f f', (∀ x, f x = f' x) → ∀ e, eval e f = eval e f'.
```

This module is instantiated twice in the choreography type: with arguments `Expr`, `Var`, `Val` and `Val`, evaluating expressions to values, and with arguments `BExpr`, `Var`, `Val` and `bool`, evaluating Boolean expressions to Booleans.

2.2 Syntax

We present the Coq definition of the type of choreographies, and afterwards briefly explain each constructor and its pretty-printing.

```
Inductive Label : Type :=
 | left : Label
 | right : Label.

Inductive Eta : Type :=
 | Com : Pid → Expr → Pid → Var → Eta
 | Sel : Pid → Pid → Label → Eta.

Inductive Choreography : Type :=
 | Interaction : Eta → Choreography → Choreography
 | Cond : Pid → BExpr → Choreography → Choreography → Choreography
 | Call : RecVar → Choreography
 | RT_Call : RecVar → (list Pid) → Choreography → Choreography
 | End : Choreography.

Definition DefSet := RecVar → (list Pid)*Choreography.

Record Program : Type :=
  { Procedures : DefSet;
    Main : Choreography }.
```

Constructor `Interaction` builds a choreography that starts with a communication (`Eta`), which can be either a value communication (`Com`) or a label selection (`Sel`). These choreographies are written as `p#e⟶ q$x;;C` or `p⟶ q[l];;C`, respectively (in general, `eta;; C` is a choreography whenever `eta:Eta`). Label selections were inherited by choreographies from linear logic and behavioural types: they are used to communicate a choice made by the sender to the receiver. In minimalistic theories of choreographies and behavioural types, it is common to restrict the set of labels that can be communicated (`Label`) to two elements, generically called `left` and `right` [8, 4]. These labels are typically used to propagate information about the local evaluation of a conditional expression, which generates two possible execution branches.

A choreography that starts by locally evaluating an expression is built using `Cond`, written `If p??b Then C1 Else C2`, while invoking procedure `X` is built as `Call X`. A procedure may involve several processes; the auxiliary term `RT_Call X ps C` represents a procedure that has already started executing, but the processes in `ps` have not yet entered it – the term `C` is obtained from the actions of procedure that have been executed (see the semantics below). We remark that `ps` has type `list Pid`, but it is interpreted as a set (in particular, it is always manipulated by means of set operations). `End` denotes the terminated choreography.

A `DefSet` (set of procedure definitions) is a mapping assigning to each `RecVar` a list of processes and a choreography; intuitively, the list of processes contains the processes that are used in the procedure. A `Program` is a pair containing a set of procedure definitions and the choreography to be executed at the start.

Terms built using `RT_Call` are meant to be runtime terms, generated while executing a choreography; therefore, programs written by programmers should not contain such terms.

We call such a choreography `initial`, and define it inductively in the natural way.

Well-formedness. There are a number of well-formedness requirements on choreographies. Some of these come from practical motivations and are typically explicitly required, while others are more technical and not always written down in other articles. We summarise these conditions.

A choreography is well-formed if its processes do not self-communicate: the two arguments of type `Pid` to `Com` and `Sel` in all its subterms are always distinct. Furthermore, the list of process names in the argument of `RT_Call` is never empty. These conditions are defined separately by recursion in the natural way.

For a program `P` to be well-formed, there are more requirements on procedure definitions and the annotations of the runtime terms. First, both `Main P` and all choreographies in `Procedures P` must be well-formed, and furthermore the latter must all be initial choreographies. In `Main P`, all runtime terms must be consistently annotated: the set `ps` in `RT_Call X ps C` should only contain processes that occur in the definition of `X` (as specified in `Procedures P`).

Second, a program must be finitely specifiable. Instead of requiring the type `RecVar` to be finite, which would significantly complicate the formalisation, we require that there exist `Xs:list RecVar` such that every procedure in `Xs`, as well as `Main P`, only calls procedures in `Xs`. Hence, it becomes irrelevant what the remaining procedure definitions are. Well-formedness is thus parameterised on `Xs`. (`Vars P X` and `Procs P X` denote the first and second components of `Procedures P X`, respectively, and `within_Xs Xs C` holds if choreography `C` only uses procedure names in `Xs`.)

```
Definition Program_WF (Xs:list RecVar) (P:Program) : Prop :=
  Choreography_WF (Main P) ∧ within_Xs Xs (Main P) ∧ consistent (Vars P) (Main P) ∧
  ∀ X, In X Xs → Choreography_WF (Procs P X) ∧ initial (Procs P X) ∧
           (Vars P X) ≠ nil ∧ within_Xs Xs (Procs P X).
```

The third and last condition is that the set of procedure definitions in a program must be well-annotated: if `Procedures P X=(ps,C)`, then the set of processes used in `C` must be included in `ps`. The set of processes used in `C` is in turn recursively defined using the information in `Procedures P`, so computing an annotation is not straightforward. For this reason, it can be convenient in practice to over-annotate a program – which is why well-annotation only requires a set inclusion. (Function `CCC_pn` computes the set of processes occurring in a choreography, given the set of processes used in each procedure.)

```
Definition well_ann (P:Program) : Prop :=
  ∀ X, set_incl _ (CCC_pn (Procs P X) (Vars P)) (Vars P X).

Definition CCP_WF (P:Program) := well_ann P ∧ ∃ Xs, Program_WF Xs P.
```

While `Program_WF` is decidable, `well_ann` and `CCP_WF` (for CC program) are not, due to the quantifications in their definitions. This motivates defining `Program_WF` separately.

Formalising well-formedness requires some auxiliary definitions (the sets of processes and procedure names used in a choreography) and several inductive definitions. Most of them are straightforward, if sometimes cumbersome; the complexity of the final definition can make proofs of well-formedness quickly grow in size and number of cases, so we provide a number of inversion results such as the following to make subsequent proofs easier.

```
Lemma CCP_WF_eta : ∀ Defs C eta,
  CCP_WF (Build_Program Defs (eta;;C)) → CCP_WF (Build_Program Defs C).
```

2.3 Semantics

The semantics of CC is defined by means of labelled transition systems, in several layers. At the lowest layer, we define the transitions that a choreography can make (`CCC_To`), parameterised on a set of procedure definitions; then we pack these transitions into the more usual presentation – as a labelled relation `CCP_To` on *configurations* (pairs program/state). Finally, we define multi-step transitions `CCP_ToStar` as the transitive and reflexive closure of the transition relation. This layered approach makes proofs about transitions cleaner, since the different levels of induction are separated.

Transition labels. We have two types of transition labels. The first one is a simple inductive type with constructors corresponding to the possible actions a choreography can take: value communications, label selections, local conditional, or local procedure call. This type is called `RichLabel`: rich labels are not present in the informal theory [29], but they are needed to obtain strong enough induction hypotheses in proofs of results about CC that we will need in Section 4 for Turing completeness. The labels in the informal theory correspond to observable actions; they are formalised as `TransitionLabel`, and they forget the internal details of actions. The two types are connected by a function `forget:RichLabel → TransitionLabel`.

```
Inductive RichLabel : Type :=
| R_Com (p:Pid) (v:Value) (q:Pid) (x:Var) : RichLabel
| R_Sel (p:Pid) (q:Pid) (l:Label) : RichLabel
| R_Cond (p:Pid) : RichLabel
| R_Call (X:RecVar) (p:Pid) : RichLabel.

Inductive TransitionLabel : Type :=
| L_Com (p:Pid) (v:Value) (q:Pid) : TransitionLabel
| L_Sel (p:Pid) (q:Pid) (l:Label) : TransitionLabel
| L_Tau (p:Pid) : TransitionLabel.
```

Transition relations. `CCC_To` is defined inductively by a total of 11 clauses, corresponding to the 11 rules in the informal presentation. We include some of them below, with some proof terms omitted.

```
Inductive CCC_To (Defs : DefSet) :
  Choreography → State → RichLabel → Choreography → State → Prop :=
 | C_Com p e q x C s s' : let v := (eval_on_state e s p) in
       eq_state_ext s' (update s q x v) →
       CCC_To Defs (p #e ⟶ q$x;; C) s (R_Com p v q x) C s'
 | C_Delay_Eta eta C C' s s' t: disjoint_eta_rl eta t →
       CCC_To Defs C s t C' s' →
       CCC_To Defs (eta;; C) s t (eta;; C') s'
 | C_Call_Start p X s s':
       eq_state_ext s s' →
       set_size _ (fst (Defs X)) > 1 → In p (fst (Defs X)) →
       CCC_To Defs
              (Call X) s
              (R_Call X p)
              (RT_Call X (set_remove _ p (fst (Defs X))) (snd (Defs X))) s'
 | C_Call_Enter p ps X C s s':
       eq_state_ext s s' → set_size _ ps > 1 → In p ps →
       CCC_To Defs
              (RT_Call X ps C) s
```

```
            (R_Call X p)
            (RT_Call X (set_remove _ p ps) C) s'
| C_Call_Finish p ps X C s s':
      eq_state_ext s s' → set_size _ ps = 1 → In p ps →
      CCC_To Defs
            (RT_Call X ps C) s (R_Call X p) C s'.
```

The first constructor defines a transition for a value communication, with the state being updated with the value received at the receiver. Since states are functions, we do not want to require that the resulting state be `update s q x v` – the state obtained directly from `s` by updating the value at `q`'s variable `x` with `v` – but only that it be extensionally equal to it (the values of variables at all processes are the same). (The stronger requirement would break, e.g., confluence, since updating two different variables in a different order does not yield the same state.)

The original informal theory allows for out-of-order execution of independent interactions [8, 29], a well-established feature of choreographic languages [6, 19]. For example, given a choreography that consists of two independent communications such as `p#e⟶ q$x;;r#e'⟶ s$y;;End` ("`p` communicates `e` to `q` and `r` communicates `e'` to `s`") where `p`, `q`, `r`, and `s` are distinct processes, we should be able to observe the first and the second value communication in whichever order. Out-of-order execution is modelled by three rules, of which the second constructor shown is an example. Here, a choreography is allowed to reduce under a prefix `eta` if its label does not share any processes with `eta`. This side-condition is checked by `disjoint_eta_rl eta t`; several auxiliary predicates named `disjoint_type1_type2` are defined to simplify writing these conditions.

Procedure calls are managed by four rules, of which the main three are shown. A procedure call is expanded when the first process involved in it enters it (rule `C_Call_Start`). The remaining processes and the procedure's definition are stored in a runtime term, from which we can observe transitions either by more processes entering the procedure (rule `C_Call_Enter`) or by out-of-order execution of internal transitions of the procedure (rule `C_Delay_Call`, not shown). When the last process enters the procedure, the runtime term is consumed (rule `C_Call_Finish`). The missing rule addresses the edge case when a procedure only uses one process.

In order to prove results about transitions, it is often useful to infer the resulting choreography and state. The constructors of `CCC_To` cannot be used for this purpose, since the resulting state is not uniquely determined. Therefore, we prove a number of lemmas stating restricted forms of transitions that are useful for forward reasoning.

```
Lemma C_Com' : ∀ Defs p e q x C s, let v := (eval_on_state e s p) in
      CCC_To Defs (p #e ⟶ q$x;; C) s (R_Com p v q x) C (update s q x v).
```

Afterwards, we formalise the transition relations as defined in [29].

```
Definition Configuration : Type := Program *State.

Inductive CCP_To : Configuration → TransitionLabel → Configuration → Prop :=
 | CCP_To_intro Defs C s t C' s' : CCC_To Defs C s t C' s' →
    CCP_To (Build_Program Defs C,s) (forget t) (Build_Program Defs C',s').

Inductive CCP_ToStar : Configuration → list TransitionLabel → Configuration → Prop :=
 | CCT_Refl c : CCP_ToStar c nil c
 | CCT_Step c1 t c2 l c3 : CCP_To c1 t c2 → CCP_ToStar c2 l c3 → CCP_ToStar c1 (t::l) c3.
```

We also define the suggestive notations `c —[tl ]⟶ c'` for `CCP_To c tl c'` and `c —[ts ]⟶ *c'` for `CCP_To_Star c ts c'`.

2.4 Progress, Determinism, and Confluence

The challenging – and interesting – part of formalising CC is establishing the basic properties of the language, which are essential for more advanced results and typically not proven in detail in publications. We discuss some of the issues encountered, as these were the driving force behind the changes relative to [8].

The first key property of choreographies is that they are deadlock-free by design: any choreography that is not terminated can reduce. Since the only terminated choreography in CC is `End`, this property also implies that any choreography either eventually reaches the terminated choreography `End` or runs infinitely. These properties depend on the basic result that transitions preserve well-formedness.

```
Lemma CCC_ToStar_CCP_WF : ∀ P s l P' s', CCP_WF P → (P,s) —[l]⟶ * (P',s') → CCP_WF P'.

Theorem progress : ∀ P, Main P ≠ End → CCP_WF P → ∀ s, ∃ tl c', (P,s) —[tl]⟶ c'.

Theorem deadlock_freedom : ∀ P, CCP_WF P →
  ∀ s ts c', (P,s) —[ts]⟶ * c' → {Main (fst c') = End} + {∃ tl c'', c' —[tl]⟶ c''}.
```

This is the first place where we benefit from the change in both the syntax and semantics of choreographies from [8] to [29], which removes idiosyncrasies that required clarifications in the reviewing process of [8]. In the language of [8], procedures are defined inside choreographies by means of a $\mathsf{def}\ X = C_X\ \mathsf{in}\ C$ constructor in choreographies. This makes the definition of terminated choreography much more complicated, since `End` could occur inside some of these terms. Furthermore, procedure calls were expanded by structural precongruence, so that a choreography as $\mathsf{def}\ X = \mathsf{End}\ \mathsf{in}\ X$ would also be terminated. Separating procedure definitions from the main choreography in a program and promoting procedure calls to transitions makes stating and proving progress much simpler. The direct syntactic characterisation of termination also has advantages, since it is intuitive and easily verifiable.

The second key property is confluence, which is an essential ingredient of the proof of Turing completeness below: if a choreography has two different transition paths, then these paths either end at the same configuration, or both resulting configurations can reach the same one. This is proved by first showing the diamond property for choreography transitions, then lifting it to one-step transitions, and finally applying induction.

```
Lemma diamond_Chor : ∀ Defs C s tl1 tl2 C1 C2 s1 s2,
  CCC_To Defs C s tl1 C1 s1 → CCC_To Defs C s tl2 C2 s2 →
  tl1 ≠ tl2 → ∃ C' s', CCC_To Defs C1 s1 tl2 C' s' ∧ CCC_To Defs C2 s2 tl1 C' s'.

Lemma diamond_1 : ∀ c tl1 tl2 c1 c2,
  c —[tl1 ]⟶ c1 → c —[tl2 ]⟶ c2 →
  tl1 ≠ tl2 → ∃ c', c1 —[tl2 ]⟶ c' ∧ c2 —[tl1 ]⟶ c'.

Lemma diamond_4 : ∀ P s tl1 tl2 P1 s1 P2 s2,
  (P,s) —[tl1 ]⟶ *(P1,s1) → (P,s) —[tl2 ]⟶ *(P2,s2) →
  (∃ P' tl1' tl2' s1' s2',
    (P1,s1) —[tl1' ]⟶ *(P',s1') ∧ (P2,s2) —[tl2' ]⟶ *(P',s2') ∧ eq_state_ext s1' s2').
```

As an important consequence, we get that any two executions of a choreography that end in a terminated choreography must yield the same state.

```
Lemma termination_unique : ∀ c tl1 c1 tl2 c2,
  c —[tl1]⟶* c1 → c —[tl2]⟶* c2 →
  Main (fst c1) = End → Main (fst c2) = End → eq_state_ext (snd c1) (snd c2).
```

The complexity of the proof of confluence was the determining factor for deciding to start our work from the variation of the choreographic language presented in [29] instead of that in [8]. The current proof of confluence takes about 300 lines of Coq code, including a total of 11 lemmas. This is in stark contrast with the previous attempt, which already included over 30 lemmas with extremely long proofs. The reason for this complexity lay, again, in both inlined procedure definitions and structural precongruence. Inlined procedure definitions forced us to deal with all the usual problems regarding bound variables and renaming (e.g. dealing with capture-avoiding substitutions, working up to α-renaming); structural precongruence introduced an absurd level of complexity because it allowed choreographies to be rewritten arbitrarily.

To understand this issue, consider again the choreography `p#e⟶ q$x;; r#e'⟶ s$y;; End`. As we have previously discussed, this choreography can execute first either the communication between `p` and `q` or the one between `r` and `s`. In our framework, the first transition is modelled by rule `C_Com`, while the second is obtained by applying rule `C_Delay_Eta` followed by `C_Com`. In a framework with reductions and structural precongruence, instead, the second transition is modelled by first rewriting the choreography as `r#e'⟶ s$y;; p#e⟶ q$x;; End` and then applying rule `C_Com` [8]. The set of legal rewritings is formally defined by the structural precongruence relation $\preceq$, and there is a rule in the semantics allowing C_1 to reduce to C_2 whenever $C_1 \preceq C_1'$, $C_2' \preceq C_2$, and C_1' reduces to C_2. Thus, the proof of confluence also needs to take into account all the possible ways into which choreographies may be rewritten in a reduction. In a proof of confluence, where there are two reductions, there are four possible places where choreographies are rewritten; given the high number of rules defining structural precongruence, this led to an explosion of the number of cases. Furthermore, induction hypotheses typically were not strong enough, and we were forced to resort to complicated auxiliary notions such as explicitly measuring the size of the derivation of transitions, and proving that transitions could be normalised. This process led to a seemingly ever-growing number of auxiliary lemmas that needed to be proved, and after several months of work with little progress it became evident that the problem lay in the formalism.

With the current definitions, the theory of CC is formalised in two files. The first file, which defines the preliminaries, contains 40 definitions, 58 lemmas and around 700 lines of code. The second file, which defines CC-specific results, contains 48 definitions, 106 lemmas, 2 theorems and around 2100 lines of code.

3 Partial Recursive Functions

In order to formalise Turing completeness of our choreographic language, we need a model of computation. In [8], the model chosen was Kleene's partial recursive functions [22], and the proof proceeds by showing that these can all be implemented as a choreography, for a suitable definition of implementation. This proof structure closely follows that of the original proof of computational completeness for Turing machines [33].

In this section, we describe our formalisation of partial recursive functions, and the main challenges and design options that it involved. Following standard pratice, we routinely use lambda notation for denoting these functions.

3.1 Syntax

The class of partial recursive functions is defined inductively as the smallest class containing the constant unary zero function $Z = \lambda x.0$, the unary successor function $S = \lambda x.x + 1$ and the n-ary projection functions $P_k^n = \lambda x_1 \dots x_n.x_k$ (base functions), and closed under the operations of composition, primitive recursion, and minimisation. All functions have an arity (natural number); the arity of Z and S is 1, and the arity of P_k^n is n. Given a function g of arity m and m functions $f_1, \dots, f_m$ of arity k, then the composition $C(g, \vec{f})$ has arity k; if g has arity k and h has arity $k + 2$, then function $R(g, h)$ defined by primitive recursion from g and h has arity $k + 1$; and if h has arity $k + 1$, then its minimisation $M(h)$ has arity k.

We formalise this class as a dependent inductive type `PRFunction` taking the arity of the function as a parameter. In order to ensure the correct number of arguments in composition, we require $f_1, \dots, f_m$ to be given as a vector of length m (the type of vectors of length m whose elements have type `A` is written in Coq as `t A m`).

```
Inductive PRFunction : nat → Set :=
  | Zero : PRFunction 1
  | Successor : PRFunction 1
  | Projection : ∀ {m k:nat}, k < m → PRFunction m
  | Composition : ∀ {k m:nat} (g:PRFunction m) (fs:t (PRFunction k) m), PRFunction k
  | Recursion : ∀ {k:nat} (g:PRFunction k) (h:PRFunction (2+k)), PRFunction (1+k)
  | Minimisation : ∀ {k:nat} (h:PRFunction (1+k)), PRFunction k.
```

Note the required proof term on the constructor for projections. The parameter `k` is one unit lower than the parameter k in the mathematical definition, since Coq's natural numbers start at 0 – this choice simplifies the development.

3.2 Semantics

A partial recursive function of arity m is meant to denote a partial function of type $\mathbb{N}^m \to \mathbb{N}$. The denotation of Z, S and P_k^n was already given above; the remaining operators are interpreted as follows, where we write $\vec{x}$ for $x_1, \dots, x_k$.

$$
\begin{aligned}
C(g, \vec{f})(\vec{x}) &= g(f_1(\vec{x}), \dots, f_m(\vec{x})) \\
R(g, h)(0, \vec{x}) &= g(\vec{x}) \\
R(g, h)(n + 1, \vec{x}) &= h(n, R(g, h)(n, \vec{x}), \vec{x}) \\
M(h)(\vec{x}) &= n \text{ if } h(\vec{x}, n) = 0 \text{ and } h(\vec{x}, i) > 0 \text{ for all } 0 \leq i < n
\end{aligned}
$$

Minimisation can lead to partiality, since there may be no n satisfying the conditions given in its definition. This partiality propagates, since any value depending on an undefined value is also undefined. Kleene's original work does not completely specify this mechanism: for example, if f is a unary function that is undefined everywhere, should $C(Z, f)$ be also everywhere undefined, or constantly zero? It is common practice to follow a call-by-value semantics and assume that a function is undefined whenever any of its arguments is undefined, even in the case where those arguments are not used for computing the result; we take this approach in this work.

Since Coq does not allow for defining partial functions, we take an operational approach to the semantics of partial recursive functions, and interactively define the bounded evaluation of an m-ary function f on a vector of length m in n steps, which has type `option nat`. This construction proceeds in three steps. First, we deal with minimisation by defining

```
Fixpoint find_zero_from {k} (h:t (option nat) (1+k) → option nat)
  (ns:t (option nat) k) (init:nat) (steps:nat) : option nat :=
  match steps with
  | O ⇒ None
  | S m ⇒ match h (shiftin (Some init) ns) with
          | None ⇒ None
          | Some O ⇒ Some init
          | Some (S _) ⇒ find_zero_from h ns (S init) m end end.
```

which tries to find the smallest zero of `h` starting at `init` using a bound of `steps` steps for the first value, `steps`−1 for the next value, etc. (The type `t T n` is the type of vectors containing exactly `n` elements of type `T`.) With this, we recursively define the evaluation function

```
Fixpoint eval_opt {m} (f:PRFunction m) : ∀ (steps:nat) (ns:t (option nat) m), option nat.
```

Defining this function directly is complex due to the dependent type `PRFunction m` – one needs to do induction on `m` and then reason about the possible cases for `f` in each case, which is not easy to write directly. Instead, we define `eval_opt` directly. Finally, we define

```
Definition eval {m} (f:PRFunction m) (steps:nat) (ns:t nat m) : option nat
  := eval_opt f steps (map Some ns)
```

as our intended evaluation function.

Evaluation starts by checking that all arguments are defined. If this is the case, then the base functions always return their value; composition and recursion call the functions that they depend upon with the same number of steps; and minimisation initiates a search from 0 with the bounds explained above.

In order to ensure that the interactive definitions are correct, we prove a number of lemmas stating that the defining equations for each class of functions hold. For example, for recursion we have the following three lemmas.

```
Lemma Recursion_correct_base : ∀ k (g:PRFunction k) (h:PRFunction (2+k)) (ns:t nat (1+k)),
  ∀ steps, hd ns = 0 → eval (Recursion g h) steps ns = eval g steps (tl ns).

Lemma Recursion_correct_step : ∀ k (g:PRFunction k) (h:PRFunction (2+k)) (ns:t nat (1+k)),
  ∀ steps x y, hd ns = S x → (eval (Recursion g h) steps (x :: tl ns)) = Some y →
  eval (Recursion g h) steps ns = eval h steps (x :: y :: tl ns).

Lemma Recursion_correct_step' : ∀ k (g:PRFunction k) (h:PRFunction (2+k)) (ns:t nat (1+k)),
  ∀ steps x, hd ns = S x → (eval (Recursion g h) steps (x :: tl ns)) = None →
  eval (Recursion g h) steps ns = None.
```

These results rely on a number of auxiliary results, notably about the function `find_zero_from`.

3.3 Examples

For further proof of correctness, we chose some functions that typically are used as examples in textbooks on the topic – addition, multiplication, sign – and some relations – greater than, smaller than, equals – and showed that the usual definitions are correct. For example, sum is defined as $R(P^1_1, C(S, P^3_2))$ (this is also used as an example in [8]). Our formalisation defines `PR_add` as `Recursion (Projection aux11) (Composition Successor [Projection aux23])`, and includes

```
Lemma add_correct : ∀ m n steps, eval PR_add steps [m; n] = Some (m + n).
```

(The parameters m and k are implicit in the constructor for projections; the proof terms are named to correspond to the informal usage, so that `aux11`:0<1. Thus P_1^1 is represented by `@Projection 0 1 aux11`.)

3.4 Convergence and Uniqueness

The next step is to show that the value by `eval` is unique and stable (augmenting the number of steps can only change it from `None` to `Some n`, and not conversely).

This is the first place where we have to do induction over `PRFunction`. The induction principle automatically generated by Coq from the type definition is not strong enough for our purposes: the constructor for composition includes elements of type `PRFunction` inside a vector argument, and these functions are not available on inductive proofs. We use a standard technique to overcome this limitation: we assign a depth to every element of `PRFunction` (corresponding to the depth of its abstract syntax tree), and prove results by induction over the depth of functions. In particular, this allows us to prove the following general induction principle.

```
Theorem PRFunction_induction : ∀ (P:∀ (n:nat) (f:PRFunction n), Prop),
  P _ Zero → P _ Successor →
  (∀ i j (Hp:i<j), P _ (Projection Hp)) →
  (∀ m k g fs, (∀ H, P m fs[@H]) → P k g → P _ (Composition g fs)) →
  (∀ k g h, P _ g → P _ h → P (1+k) (Recursion g h)) →
  (∀ k h, P _ h → P k (Minimisation h)) →
  ∀ n f, P n f.
```

(The notation `v[@H]` denotes the k-th element of vector `v`, where `H` is a proof that k is smaller than the length of `v`.)

Using this principle (and sometimes directly induction over depth), we can prove all mentioned properties of evaluation. We then define convergence and divergence in the natural way.

```
Definition converges {k} (f:PRFunction k) ns y := ∃ steps, eval f steps ns = Some y.
Definition diverges {k} (f:PRFunction k) ns := ∀ steps, eval f steps ns = None.
Lemma converges_inj : ∀ {k} f ns y y', converges (k:=k) f ns y → converges f ns y' → y = y'.
Lemma converges_diverges : ∀ {k} f ns, (diverges (k:=k) f ns ↔ ∀ y, ~converges f ns y).
```

Finally, we prove a number of results for establishing convergence of each class of functions. These results are used later, when proving Turing completeness of choreographies. For example, for recursion we have:

```
Lemma Recursion_converges_base : ∀ k g h ns y,
  converges g (tl ns) y → converges (@Recursion k g h) (0::tl ns) y.

Lemma Recursion_converges_step : ∀ k g h ns x y z,
  converges (@Recursion k g h) (x::ns) y →
  converges h (x::y:: ns) z → converges (Recursion g h) (S x::ns) z.
```

Conversely, if recursion converges, then all intermediate computations must also converge.

```
Lemma converges_Recursion_base : ∀ {m} (g:PRFunction m) h ns y,
  converges (Recursion g h) ns y → hd ns = 0 → converges g (tl ns) y.

Lemma converges_Recursion_step : ∀ {m} (g:PRFunction m) h ns x y,
  converges (Recursion g h) ns y → hd ns = (S x) →
```

```
  ∃ z, converges (Recursion g h) (x :: tl ns) z ∧ converges h (x :: z ::  tl ns) y.

Lemma converges_Recursion_full : ∀ {m} (g:PRFunction m) h ns y,
  converges (Recursion g h) ns y →
  ∀ x, x ≤ hd ns → ∃ z, converges (Recursion g h) (x :: tl ns) z.
```

For completeness, the formalisation also includes corresponding results for divergence; these are currently unused.

This part of the development contains 22 definitions and 84 lemmas, with a total of 1388 lines of code. (This excludes some results on basic data structures that we could not find in the standard library.)

4 Turing Completeness of Choreographies

We are now ready to show that the choreographic language is Turing complete, in the sense that every partial recursive function can be implemented as a choreography (for a suitable definition of implementation). The construction is very similar to that in [8]: a significant part of the formalisation amounted to transcribing all the relevant definitions to Coq syntax. This contributes to confirming that the simplifications introduced in [29] and the additional notions and properties (rich labels, well-formedness, etc.) introduced by our formalisation are mostly internal and aimed at simplifying metatheoretical reasoning on choreographies.

4.1 Concrete Language

The first step is to instantiate the parameters in the definition of `CC` with the right types. Process identifiers, values and procedure names are natural numbers. Each process contains two variables; we use `Bool` for this type, and alias its elements to `xx` and `yy` for clarity. Expressions are an inductive type with three elements: `this` (evaluating to the process's value at `xx`), `zero` (evaluating to 0) and `succ_this` (evaluating to the successor of the value at `xx`). Boolean expressions are a singleton type with one element `compare`, which evaluates to `true` exactly when the process's two variables store the same element.

We restrict the syntax of choreographies to mimic the operators from the original development in [8], where processes only had one storage variable. Thus, incoming value communications are always stored at variable `xx`. However, in that calculus the conditional compared the value stored in two processes. We model this as a communication whose result is stored at `yy`, followed by a call to `compare`. We define these operations as macros.

```
Definition Send p e q := p#e ⟶ q$xx.
Definition IfEq p q C1 C2 := q#this ⟶ p$yy;; If p ? compare Then C1 Else C2.
```

4.2 Encoding

The most complex step of the formalisation is formalising the encoding of partial recursive functions as choreographies. This is naturally a recursive construction, but there are some challenges. First, non-base functions need to store intermediate computation results in auxiliary processes; second, recursion and minimisation use procedure definitions to implement loops. The strategy in [8] was to use auxiliary processes sequentially: since we can statically determine how many processes are needed from the definition of the function to encode, we can always determine the first unused process. The language used therein did not have the problem with recursion variables, but the same technique applies.

The key definition is the following: a choreography C implements function $f : \mathbb{N}^m \to \mathbb{N}$ with input processes $p_1, \ldots, p_m$ and output process q iff: for any state s where $p_1, \ldots, p_m$ contain the values $n_1, \ldots, n_m$ in their variable `xx`, (i) if $f(n_1, \ldots, n_m) = n$, then all executions of C from s terminate, and do so in a state where q stores n in its variable `xx`; and (ii) if $f(n_1, \ldots, n_m)$ is undefined, then execution of C from s never terminates. This is captured in the following Coq definition.

```
Definition implements (P:Program) {n} (f:PRFunction n) (ps:t Pid n) (q:Pid) :=
  ∀ (xs:t nat n) (s:State), (∀ Hi, s (ps[@Hi]) xx = xs[@Hi]) →
  (∀ y, converges f xs y ↔ ∃ s' ts P',
      (P,s) —[ts]⟶* (P',s') ∧ s' q xx = y ∧ Main P' = End) ∧
  (diverges f xs ↔ ∀ s' ts P', (P,s) —[ts]⟶* (P',s') → Main P' ≠ End).
```

The idea is that we recursively define the set of procedure definitions needed to encode $f : \mathbb{N}^m \to \mathbb{N}$, taking as parameters not only the processes $p_1, \ldots, p_m$ and q, but also the indices of the first unused process and the first unused procedure. The encoding of f is a program whose set of procedure definitions is obtained by instantiating the last two values to $\max(p_1, \ldots, p_m, q) + 1$ and 0, respectively, and whose main choreography is `Call 0`. Furthermore, we also ensure that the choreography terminates by calling the first unused procedure (which by default is defined as the terminated choreography). This makes it easy to ensure that procedure calls compose nicely in the recursive steps of the construction.

We start by defining two auxiliary functions `Pi` and `Gamma`, which given a function in `PRFunction` return the number of processes and procedures needed to encode it, respectively. (Function `Pi` is exactly the function Π from [8].) No results about these functions are needed – their definition suffices to prove all needed results, namely that no process (resp. procedure) higher than `Pi f` (resp. `Gamma f`) is used when encoding `f`.

Ironically, the most challenging part of the definition is composition (which is straightforward in the informal presentation), and not minimisation (which is responsible for introducing partiality). The base cases are directly encoded by suitably adapting the definitions from [8], and those of recursion and minimisation have a recursive structure very close to the informal textbook definition. However, the definition of composition needs to be recursive (due to the variable number of argument functions), and working with vectors adds a layer of complexity. As such, a number of auxiliary functions were defined to deal with composition, defining a choreography that encodes a vector of functions all with the same inputs and returning outputs in consecutive processes.

The recursive definition of encoding `Encoding_rec` is again written interactively, and afterwards a number of lemmas prove that it behaves as expected, e.g.:

```
Lemma Zero_Procs : ∀ d Hd ps q n X,
  Encoding_rec Zero d Hd ps q n X X = Send (hd ps) zero q;; Call (S X).

Lemma Recursion_Procs_g : ∀ k (g:PRFunction k) h d (Hd:depth (Recursion g h) < S d) ps q n,
  ∀ X Y, X ≤ Y < X + Gamma g → let Hg := (...) in
  Encoding_rec (Recursion g h) _ Hd ps q n X Y =
    Encoding_rec _ _ Hg (tl ps) n (n+3) X Y.
```

where we omit the definition of the proof term `Hg`. Note that `Encoding_rec` has the complex type `∀ (m:nat) (f:PRFunction m) (d:nat), depth f<d → t Pid m → Pid → nat → RecVar → RecVar → Choreography` – it receives a function of depth smaller than d, the set of input processes, the output process, the index of the first unused process and the index of the first unused procedure definition, and returns a mapping of procedure definitions to choreographies (where all previously defined procedures are unchanged).

Finally, we prove that encoding always returns a well-formed choreography. This is implicit in [8], but it is an essential property that should hold. For convenience, each condition of well-formedness is proved separately, capitalising on the fact that the encoding returns an initial choreography. The proof follows the recursive structure of the definition of `Encoding_rec`, and is relatively automatic once the relevant splitting in cases is done.

```
Lemma Encoding_WF : ∀ {n} (f:PRFunction n) ps q, ~In q ps → CCP_WF (Encoding f ps q).
```

4.3 Soundness

Soundness of the encoding – the property that the encoding of f implements f – is proven by analysing the execution path obtained by always reducing the first action in the choreography, and invoking confluence. We split the proof into a number of lemmas, stating the obvious reductions from each procedure definition to the procedure call at its end. We give some examples of these results.

```
Lemma Zero_reduce : ∀ Defs (ps: t Pid 1) q X s,
  ∃ t, (Build_Program Defs (Send (hd ps) zero q;; Call X),s)
    —[t]⟶ (Build_Program Defs (Call X), update s q xx 0).

Lemma Recursion_reduce_0 : ∀ Defs X n s,
  ∃ t, (Build_Program Defs (Send (n + 2) zero (S n);; Call X),s)
    —[t]⟶ (Build_Program Defs (Call X),update s (S n) xx 0).

Lemma Recursion_reduce_1_true : ∀ m Defs X Y n (ps:t Pid (S m)) q s,
  s (S n) xx = s (hd ps) xx → ∃ t s',
  (Build_Program Defs (IfEq (S n) (hd ps) (Send n this q;; Call X) (Call Y)),s)
    —[t]⟶* (Build_Program Defs (Call X), s')
  ∧ s' q xx = s n xx ∧ ∀ p, p ≠ q → s' p xx = s p xx.

Lemma Recursion_reduce_1_false : ∀ m Defs X Y n (ps:t Pid (S m)) q s,
  s (S n) xx ≠ s (hd ps) xx → ∃ t,
  (Build_Program Defs (IfEq (S n) (hd ps) (Send n this q;; Call X) (Call Y)),s)
    —[t]⟶* (Build_Program Defs (Call Y), update s (S n) yy (s (hd ps) xx)).
```

We briefly explain the lemmas about recursion. Encoding $R(g,h)$ uses three auxiliary procedures. The first one initializes the recursion by placing a zero on the first auxiliary process (Lemma `Recursion_reduce_0`), and calls the first procedure in the encoding of g. The second one, placed immediately after the procedures used for encoding g, checks whether the value in the auxiliary process is the value where we want to stop, and in this case places the result in the return process (Lemma `Recursion_reduce_1_true`). Otherwise, it calls the first procedure in the encoding of h (Lemma `Recursion_reduce_1_true`). (This explanation assumes the values of `X` and `Y` to be instantiated in the right way, but the lemmas do not depend on this.) The third procedure, invoked when h terminates, increases the value of the auxiliary process controlling the loop (Lemma `Recursion_reduce_2`, omitted). All these lemmas also include characterisations of the resulting state that are needed for applying the induction hypothesis in the main proof.

Using these results, we can prove by induction that, if s is a state where $n_1, \ldots, n_m$ are stored in the appropriate processes, (i) if $f(n_1, \ldots, n_k)$ converges, then there is some execution path of the encoding of f from s that terminates with the expected result; (ii) using confluence, all execution paths from s must terminate in the same state; and (iii) that if $f(n_1, \ldots, n_k)$ diverges, then no execution of the encoding of f from s terminates. These

three results are combined in a single theorem, stating that the default encoding of f (where $n_1, \ldots, n_m$ are stored in processes $1, \ldots, m$, and the result is returned in process 0) is sound.

```
Theorem encoding_sound : ∀ n (f:PRFunction n), implements (Encoding' f) f (vec_1_to_n n) 0.
```

This part of the development contains 28 definitions and 65 lemmas, with a total of 2352 lines of code.

5 Discussion

We presented a formalisation of a choreographic language and proved it Turing complete. To the best of our knowledge, this is the first time that such a task has been achieved – the only comparable work in progress consists of a preliminary presentation on a certified compiler from choreographies to CakeML [18], which however does not deal with the major challenge of recursion (the choreography language used therein is simplistic and can only express finite behaviour) [24]. Moreover, we showed how formalising proofs unveiled subtle problems in definitions and can influence the development of the theory, making a case for a more systematic use of theorem provers in research in the field. The number of choreographic languages proposed in the literature is increasing rapidly, to include features of practical value such as asynchronous communication, non-determinism, broadcast, dynamic network topologies, and more [2, 15, 20]. Hopefully, our work can contribute a solid foundation for the development of these features. This recalls the situation found in the field of process calculi, and indeed similar conclusions are drawn in an article that presents the formalisation of a higher-order process calculus in Coq [27].

Our formalisation of Kleene's partial recursive functions is similar to the one in [35], of which we were not aware when we started this project. The library of proofs of undecidability formalised in Coq, being developed at the University of Saarbrücken, also includes definitions and results related to ours [13, 14].

It is interesting that the proof of Turing completeness – both construction and proofs – still closely follows the original theory [8], despite the significant changes to the lowest layers that we had to make. This suggests that the major formalisation challenge currently lies in the foundational work.

Our formalisation includes some design options. The most significant one, in our opinion, is the restriction to only two labels in selections. However, as is well known in the field of session types, this is not a serious restriction [4]. Labels are typically used to communicate choices based on a conditional; more complex decisions are expressed as nested conditionals, and can be communicated by sending multiple label selections.

Restricting the set of labels to two elements also has a strong impact on the formalisation of realisability [3, 5], which we do not discuss in this article. Choreography realisability deals with identifying sufficient conditions for a choreography to be implementable in a distributed setting, and generating an implementation in a process calculus automatically. The formalisation of this construction, described in [10], was completed after the current work, and heavily relies on the set of labels being fixed. An important result is that any choreography can be amended into a realisable one, so that in particular our Turing-completeness result immediately implies Turing-completeness of the process calculus used for implementations.

We aimed at making our development reusable, so that it can readily be extended to more expressive choreographic languages. In the future, we plan to look at interesting extensions (such as those mentioned above) and explore how easy it is to extend the current formalisation to those frameworks. We conjecture that this will prove much simpler than the current effort, thanks to the structures established in this work.

References

1 Elvira Albert and Ivan Lanese, editors. *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*. Springer, 2016.

2 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2–3):95–230, 2016.

3 Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. In John Field and Michael Hicks, editors, *Procs. POPL*, pages 191–202. ACM, 2012. doi:10.1145/2103656.2103680.

4 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In Paul Gastin and François Laroussinie, editors, *Procs. CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010. doi:10.1007/978-3-642-15375-4_16.

5 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8:1–8:78, 2012. doi:10.1145/2220365.2220367.

6 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *Procs. POPL*, pages 263–274. ACM, 2013. doi:10.1145/2429069.2429101.

7 Luís Cruz-Filipe and Fabrizio Montesi. Choreographies in practice. In Albert and Lanese [1], pages 114–123. doi:10.1007/978-3-319-39570-8_8.

8 Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. *Theor. Comput. Sci.*, 802:38–66, 2020. doi:10.1016/j.tcs.2019.07.005.

9 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Choreographies in Coq. In *TYPES 2019, Abstracts*, 2019. Extended abstract.

10 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Certifying choreography compilation. *CoRR*, abs/2102.10698, 2021. URL: https://arxiv.org/abs/2102.10698.

11 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. A formalisation of a Turing-complete choreographic language in Coq, January 2021. doi:10.5281/zenodo.4479102.

12 Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Log. Methods Comput. Sci.*, 13(2), 2017. doi:10.23638/LMCS-13(2:1)2017.

13 Yannick Forster, Dominik Kirst, and Gert Smolka. On synthetic undecidability in Coq, with an application to the Entscheidungsproblem. In Assia Mahboubi and Magnus O. Myreen, editors, *Procs. CPP*, pages 38–51. ACM, 2019. doi:10.1145/3293880.3294091.

14 Yannick Forster, Fabian Kunze, and Maximilian Wuttke. Verified programming of Turing machines in Coq. In Jasmin Blanchette and Cătălin Hriţcu, editors, *Procs. CPP*, pages 114–128. ACM, 2020. doi:10.1145/3372885.3373816.

15 Simon J. Gay, Vasco T. Vasconcelos, Philip Wadler, and Nobuko Yoshida. Theory and applications of behavioural types (Dagstuhl Seminar 17051). *Dagstuhl Reports*, 7(1):158–189, 2017. doi:10.4230/DagRep.7.1.158.

16 Saverio Giallorenzo, Ivan Lanese, and Daniel Russo. Chip: A choreographic integration process. In Hervé Panetto, Christophe Debruyne, Henderik A. Proper, Claudio Agostino Ardagna, Dumitru Roman, and Robert Meersman, editors, *Procs. OTM, part II*, volume 11230 of *Lecture Notes in Computer Science*, pages 22–40. Springer, 2018. doi:10.1007/978-3-030-02671-4_2.

17 Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choreographies as objects. *CoRR*, abs/2005.09520, 2020. URL: https://arxiv.org/abs/2005.09520.

18 Alejandro Gomez-Londono and Johannes Aman Pohjola. Connecting choreography languages with verified stacks. In *Procs. of the Nordic Workshop on Programming Theory*, pages 31–33, 2018. URL: http://hdl.handle.net/102.100.100/86327?index=1.

19 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9, 2016. Also: POPL, pages 273–284, 2008. doi:10.1145/2827695.

20 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.

21 Intl. Telecommunication Union. Recommendation Z.120: Message Sequence Chart, 1996.

22 Stephen Cole Kleene. *Introduction to Metamathematics*, volume 1. North-Holland Publishing Co., 1952.

23 Alberto Lluch-Lafuente, Flemming Nielson, and Hanne Riis Nielson. Discretionary information flow control for interaction-oriented specifications. In Narciso Martí-Oliet, Peter Csaba Ölveczky, and Carolyn L. Talcott, editors, *Logic, Rewriting, and Concurrency*, volume 9200 of *Lecture Notes in Computer Science*, pages 427–450. Springer, 2015. doi:10.1007/978-3-319-23165-5_20.

24 Alejandro Gómez Londoño. Choreographies and cost semantics for reliable communicating systems. Licentiate thesis, Chalmers University of Technology, 2020.

25 Hugo A. López and Kai Heussen. Choreographing cyber-physical distributed control systems for the energy sector. In Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng, editors, *Procs. SAC*, pages 437–443. ACM, 2017. doi:10.1145/3019612.3019656.

26 Hugo A. López, Flemming Nielson, and Hanne Riis Nielson. Enforcing availability in failure-aware communicating systems. In Albert and Lanese [1], pages 195–211. doi:10.1007/978-3-319-39570-8_13.

27 Petar Maksimovic and Alan Schmitt. HOCore in Coq. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2015. doi:10.1007/978-3-319-22102-1_19.

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

29 Fabrizio Montesi. Introduction to Choreographies. Accepted for publication by Cambridge University Press, 2020.

30 Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978. doi:10.1145/359657.359659.

31 Object Management Group. Business Process Model and Notation. http://www.omg.org/spec/BPMN/2.0/, 2011.

32 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019. doi:10.1145/3290343.

33 Alan M. Turing. Computability and λ-definability. *J. Symb. Log.*, 2(4):153–163, 1937. doi:10.2307/2268280.

34 W3C. WS Choreography Description Language. http://www.w3.org/TR/ws-cdl-10/, 2004.

35 Vincent Zammit. A proof of the s-m-n theorem in coq. Technical report, The Computing Laboratory, The University of Kent, Canterbury, Kent, UK, March 1997. URL: https://kar.kent.ac.uk/21524/.