ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Department of Science

Degree in Computer Science

# Mechanized, Type-Based Enforcement of Non-Interference in Choreographic Languages

**Supervisor:**
**Saverio Giallorenzo**

**Co-Supervisor:**
**Marco Peressotti**

**Author:**
**Marco Bertoni**

---

**Graduation Session of Oct. 2025**

*La dedica*
*anche quella se vuoi*
*su più righe*

# Abstract

Abstract qui (ti consiglio di farlo alla fine)

# Contents

# Chapter 1

# Introduction

**TODO: riguarda alla fine** In this thesis we study how to enforce *non-interference* for choreographic programs using a *type-based* approach that we mechanize in a proof assistant. The central idea is to track how information may flow—both explicitly, through assignments and communications, and implicitly, through control flow—and to rule out programs whose low-observable behaviour can be influenced by high-confidential data. To this end, we design a compositional type-system indexed by a program-counter label and by an environment of security labels for variables at each process. We prove that well-typed choreographies are non-interfering with respect to a user-provided flow policy, and we mechanize the metatheory in Lean, obtaining machine-checked soundness. Along the way, we isolate a few pragmatic design choices (e.g., how to account for procedure calls and concurrency) that make the rules both precise and amenable to mechanization. Finally, we discuss limitations and possible extensions (e.g., non-determinism and richer label models).

# Chapter 2

# Background

## 2.1 Choreographies

Unless otherwise indicated, all content in this chapter is derived from: *Introduction to Choreographies [9]* by *Fabrizio Montesi*, updated with the errata corrige present in Fabrizio's website [10]. **TODO: aggiungi in qualche modo citazione a lavoro di Xuauing, visto che alla fine usi la sua versione della semantica**

Choreographies are formal descriptions of the intended collaborative behavior of processes in concurrent and distributed systems. They act as protocols that specify how different components should interact to achieve a shared goal, such as authenticating a user or completing a purchase online.
Although choreographies are written in a different style than traditional local programs, they execute like normal languages: through a sequence of transitions that represent communication and computation steps. However, instead of capturing the state of a single process, a choreography represents the global state of all participants involved, encompassing the joint execution and interactions that collectively implement the protocol.

### 2.1.1 Processes

The cornerstone of the language will be the notion of *process*. Processes are independent participants in a choreography; they can perform local computation and interact with other processes by communicating with them. From the perspective of computer systems, processes are abstract representations of computer programs executed concurrently; each process possesses its own control state and memory.
In this work, processes are usually ranged over by $p, q, r, s, \ldots$ and the infinite set of *process names* will be referred to as **PName**.

### 2.1.2 Syntax

We introduce the following language, called *Recursive Choreographies* and defined by the following context-free grammar:

$$
\begin{array}{rcl}
\mathscr{C} & ::= & \{X_i(\vec{p_i}) = C_i\}_{i \in I} \\
C & ::= & I; C \mid \mathbf{0} \\
I & ::= & p.e \to q.x \mid p \to q[L] \mid p.x := e \mid \text{if } p.e \text{ then } C_1 \text{ else } C_2 \mid X(\vec{p}\,) \mid q : X(\vec{p}\,).C \\
e & ::= & v \mid x \mid f(\vec{e}\,)
\end{array}
$$

Let us explain the grammatical entities introduced:

- $\mathscr{C}$ denotes the *context of procedure definitions*:
  A set of procedure definitions is a (possibly empty) set of equations of the form $X(\vec{p}\,) = C$[1], read *procedure $X$ has parameters $\vec{p}$ and body $C$*, where all procedure names are distinct. We call the parameters $\vec{p}$ of a procedure definition the *formal parameters* of the procedure.

- $C$ denotes a *Choreography*, which can either be a *terminated choreography* $\mathbf{0}$ (the choreography that prescribes no interactions) or the sequential composition of an *Instruction* $I$ and a *Continuation* $C$
  We denote with **Chor** the set of all entities generated by this non-terminal.

- $I$ denotes an *Instruction*, which can be one of the following:

  - A *communication* $\boldsymbol{p.e \to q.x}$, where process $p$ evaluates the *expression* $e$ locally and communicates the resulting value to process $q$ which stores it in its local variable $x$.
  - A *selection* $\boldsymbol{p \to q[L]}$. The key idea behind the introduction of label selections goes beyond the scope of this work, we will just treat it as an arbitrary instruction of the language.
  - A *local assignment* $\boldsymbol{p.x := e}$, where $p$ evaluates expression $e$ and stores the resulting value in its variable $x$.
  - A *conditional* $\textbf{if } \boldsymbol{p.e}\ \textbf{then } \boldsymbol{C_1}\ \textbf{else } \boldsymbol{C_2}$, which reads *process $p$ evaluates expression $p$, and then the choreography proceeds as $C_1$ if the result of the evaluation is the value true, or as $C_2$ otherwise.* Thus, we now assume that the set of possible values contains the Boolean value *true*. Given a conditional $\textbf{if } \boldsymbol{p.e}\ \textbf{then } \boldsymbol{C_1}\ \textbf{else } \boldsymbol{C_2}$, $e$ is called the *guard* of the conditional; and the two choreographies $C_1$ and $C_2$ are called the *branches* of the conditional or, more precisely, $C_1$ is the *then-branch* and $C_2$ is the *else-branch*.

[1] The symbol $\vec{p}$ refers to the sequence $p_1, p_2, \ldots, p_n$, consisting of the individual components $p_i$, where $n \in \mathbb{N}$ is unspecified

  - A procedure call $\boldsymbol{X(\vec{p}\,)}$, which reads *run procedure $X$ with the processes $\vec{p}$*. We call the processes $\vec{p}$ the *arguments* of the procedure call.
  - A runtime term $\boldsymbol{X(\vec{p}\,).C}$, The key idea behind the introduction of runtime terms goes beyond the scope of this work, we will just treat it as an arbitrary instruction of the language.

- $e$ denotes an *expression*, which can take three forms:
  - A *constant value* $v$. We write **Val** for the set of all possible values.
  - A *variable* $x$. We write **Var** for the set of all possible variable names.
  - A *function call* $f(\vec{e}\,)$, where $f$ is a *function name* and $\vec{e}$ are the *arguments* of $f$. A function name $f$ is a reference to a function that maps value tuples to values. The idea is that these functions can be evaluated locally, in the sense that processes compute their results without communicating with other processes. For this reason, we call functions ranged over by $f$ also *local functions*.

  We use **Expr** for the set of entities generated by $e$

### 2.1.3 Semantics

The interpretation for Choreographies is given as small-step operational semantics [14], forming a Labelled Transition System $(S, \mathbf{TLabel}, \dot{\rightarrow})$.
$S$ is the configuration space, **TLabel** represents the set of possible *transition labels* and $\dot{\rightarrow}$ represents the *transition relation*.

#### Configuration and Choreographic Store

The configurations have the form $\langle C, \Sigma, \mathscr{C} \rangle$. $C$ and $\mathscr{C}$ were defined previously in this document. $\Sigma$ represents the *Choreographic Store*. Let us now define it:

A *process store* $\sigma$ models the memory of a process, mapping variables to values. Formally, a process store is a function from variables to values:

$$\sigma : \mathbf{Var} \longrightarrow \mathbf{Val}.$$

We write **PStore** for the set of all process stores. It will often be necessary to update the content of a store, so we define a notation for that purpose. Namely, we write $\sigma[x \mapsto v]$ for the update of store $\sigma$ with the new mapping $x \mapsto v$:

$$\sigma[y \mapsto v](x) = \begin{cases} v & \text{if } x = y \\ \sigma(x) & \text{otherwise.} \end{cases}$$

A *choreographic store* (i.e. **CStore**) $\Sigma$ models the memory state of an entire system: it maps process names to their respective process stores. Formally,

$$\Sigma : \mathbf{PName} \longrightarrow \mathbf{PStore}.$$

We shall write $\Sigma[p.x \mapsto v]$ for the update of store $\Sigma$ such that the local variable $x$ of process $p$ is now mapped to $v$:

$$\Sigma[q.x \mapsto v](p) = \begin{cases} \Sigma(p)[x \mapsto v] & \text{if } p = q \\ \Sigma(p) & \text{otherwise.} \end{cases}$$

Store updates are left associative, that is:

$$\Sigma[p.x \mapsto v][q.y \mapsto u] = (\Sigma[p.x \mapsto v])[q.y \mapsto u].$$

We adopt extensional equality for both local and choreographic stores: two process stores are deemed equal if they return the same value for each variable, and two choreographic stores are considered equal if they return equal process stores for each process.

### Local Expression Evaluation

Given a **PStore** $\sigma$, expression $e$ and value $v$, the notation $\sigma \vdash e \downarrow v$ reads as *$e$ is evaluated to the value $v$ under the process store $\sigma$*. It is defined as the least relation derived by the following inference schema:

$$\sigma \vdash v \downarrow v \qquad \sigma \vdash x \downarrow \sigma(x) \qquad \frac{\sigma \vdash e_1 \downarrow v_1 \quad \cdots \quad \sigma \vdash e_n \downarrow v_n \quad \vdash f(v_1, \ldots, v_n) \downarrow v}{\sigma \vdash f(e_1, \ldots, e_n) \downarrow v}$$

We do not specify a system for deriving propositions of the kind $\vdash f(\vec{v}) \downarrow v$, since it is not important for our development: this system would depend on how functions are defined, which we choose to abstract from. Instead, we will just assume that such a system exists, and that for any $f$ and $\vec{v}$, it is always possible to derive $\vdash f(\vec{v})$ for some $v$.

### Transition Label

Given two different processes $p, q$, a selection label $L$ and a value $v$ we define **TLabel** as the set of objects generated by the following grammar:

$$\mathbf{TLabel} ::= \tau@p \mid p.v \to q \mid p \to q[L] \mid p.\text{then} \mid p.\text{else}$$

### Process Names of a Choreography

To define the *transition relation* we need to first define a few auxiliary operators. The first one is the function pn, formalizing the concept of *process names mentioned in an entity*.
We overload the pn name and define two different functions disjoint on their domain, respectively on *Choreographies* and *transition labels*. The one used at any given time will be clear from the context.

$$
\begin{aligned}
\mathrm{pn} : \mathbf{Chor} &\longrightarrow 2^{\mathbf{PName}} \\
\mathrm{pn}(\mathbf{0}) &\triangleq \emptyset \\
\mathrm{pn}(I; C) &\triangleq \mathrm{pn}(I) \cup \mathrm{pn}(C) \\
\mathrm{pn}(p.e \to q.x) &\triangleq \{p, q\} \\
\mathrm{pn}(p \to q[L]) &\triangleq \{p, q\} \\
\mathrm{pn}(p.x := e) &\triangleq \{p\} \\
\mathrm{pn}(\text{if } p.e \text{ then } C_1 \text{ else } C_2) &\triangleq \{p\} \cup \mathrm{pn}(C_1) \cup \mathrm{pn}(C_2) \\
\mathrm{pn}(X(\vec{p}\,)) &\triangleq \{\vec{p}\,\} \\
\mathrm{pn}(q : X(\vec{p}\,).C) &\triangleq \{q\}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{pn} : \mathbf{TLabel} &\longrightarrow 2^{\mathbf{PName}} \\
\mathrm{pn}(\tau@p) &\triangleq \{p\} \\
\mathrm{pn}(p.v \to q) &\triangleq \{p, q\} \\
\mathrm{pn}(p \to q[L]) &\triangleq \{p, q\} \\
\mathrm{pn}(p.\mathrm{then}) &\triangleq \{p\} \\
\mathrm{pn}(p.\mathrm{else}) &\triangleq \{p\}
\end{aligned}
$$

### Sequential Composition Operator

The operator is defined as follows, both on *Choreographies* and *Instructions*:

$$
\begin{aligned}
\mathbf{0} \fatsemi C &= C \\
(I; C') \fatsemi C &= (I \fatsemi C); (C' \fatsemi C) \\
I \fatsemi C &= \begin{cases} q : X(\vec{p}\,).(C' \fatsemi C) & \text{if } I = q : X(\vec{p}\,).C' \\ I & \text{otherwise} \end{cases}
\end{aligned}
$$

### Process substitution

The name substitution of a process $p$ is defined as:

$$p[r/s] \triangleq \begin{cases} s & \text{if } p = r \\ p & \text{otherwise} \end{cases}$$

We can now define the name substitution for *Choreographies*:

$$\begin{aligned}
\mathbf{0}[r/s] &\triangleq \mathbf{0} \\
(I; C)[r/s] &\triangleq (I[r/s]); (C[r/s]) \\
(p.e \to q.x)[r/s] &\triangleq (p[r/s].e) \to (q[r/s]).x \\
(p \to q[L])[r/s] &\triangleq (p[r/s]) \to (q[r/s])[L] \\
(p.x := e)[r/s] &\triangleq (p[r/s]).x := e \\
(\text{if } p.e \text{ then } C_1 \text{ else } C_2) &\triangleq \text{if } (p[r/s]).e \text{ then } (C_1[r/s]) \text{ else } (C_2[r/s]) \\
(X(\vec{p}\,)[r/s]) &\triangleq X(\vec{p}\,[r/s]) \\
(q : X(\vec{p}\,).C)[r/s] &\triangleq q[r/s] : X(\vec{p}\,[r/s]).(C[r/s])
\end{aligned}$$

### Transition Relation

At this point we have all the necessary machinery to define the *transition relation* $\to$ as the smallest relation derived by the following inference schemata:

$$\frac{\Sigma(p) \vdash e \downarrow v}{\langle p.x := e; C, \Sigma, \mathscr{C}\rangle \xrightarrow{\tau@p} \langle C, \Sigma[p.x \mapsto v], \mathscr{C}\rangle}\ \text{LOCAL}$$

$$\frac{\Sigma(p) \vdash e \downarrow v}{\langle p.e \to q.x; C, \Sigma, \mathscr{C}\rangle \xrightarrow{p.v \to q} \langle C, \Sigma[q.x \mapsto v], \mathscr{C}\rangle}\ \text{COM}$$

$$\frac{}{\langle p \to q[L]; C, \Sigma, \mathscr{C}\rangle \xrightarrow{p \to q[L]} \langle C, \Sigma, \mathscr{C}\rangle}\ \text{SEL}$$

$$\frac{\Sigma(p) \vdash e \downarrow \mathit{true}}{\langle \text{if } p.e \text{ then } C_1 \text{ else } C_2; C, \Sigma, \mathscr{C}\rangle \xrightarrow{p.\text{then}} \langle C_1 \fatsemi C, \Sigma, \mathscr{C}\rangle}\ \text{COND-THEN}$$

$$\frac{\Sigma(p) \vdash e \downarrow v \qquad v \neq \mathit{true}}{\langle \text{if } p.e \text{ then } C_1 \text{ else } C_2; C, \Sigma, \mathscr{C}\rangle \xrightarrow{p.\text{else}} \langle C_2 \fatsemi C, \Sigma, \mathscr{C}\rangle}\ \text{COND-ELSE}$$

$$\frac{X(\vec{q}\,) = C \in \mathscr{C} \qquad \vec{p} = p_1, \ldots, p_n \qquad i \in [1, n]}{\langle X(\vec{p}\,); C', \Sigma, \mathscr{C}\rangle \xrightarrow{\tau@p_i} \langle p_1 : X(\vec{p}\,).C'; \ldots; p_{i-1} : X(\vec{p}\,).C'; p_{i+1} : X(\vec{p}\,).C'; \ldots; p_n : X(\vec{p}\,).C'; C[\vec{q}/\vec{p}\,] \fatsemi C', \Sigma, \mathscr{C}\rangle}\ \text{CALL-FIRST}$$

$$\frac{}{\langle q : X(\vec{p}\,).C'; C, \Sigma, \mathscr{C}\rangle \xrightarrow{\tau@q} \langle C, \Sigma, \mathscr{C}\rangle}\ \text{CALL-ENTER}$$

$$\frac{\langle C, \Sigma, \mathscr{C}\rangle \xrightarrow{\mu} \langle C', \Sigma', \mathscr{C}\rangle \qquad \mathrm{pn}(I) \# \mathrm{pn}(\mu)}{\langle I; C, \Sigma, \mathscr{C}\rangle \xrightarrow{\mu} \langle I; C', \Sigma', \mathscr{C}\rangle}\ \text{DELAY}$$

$$\frac{\langle C_1, \Sigma, \mathscr{C}\rangle \xrightarrow{\mu} \langle C_1', \Sigma', \mathscr{C}\rangle \qquad \langle C_2, \Sigma, \mathscr{C}\rangle \xrightarrow{\mu} \langle C_2', \Sigma', \mathscr{C}\rangle \qquad p \notin \mathrm{pn}(\mu)}{\langle \text{if } p.e \text{ then } C_1 \text{ else } C_2; C, \Sigma, \mathscr{C}\rangle \xrightarrow{\mu} \langle \text{if } p.e \text{ then } C_1' \text{ else } C_2'; C, \Sigma', \mathscr{C}\rangle}\ \text{DELAY-COND}$$

Let us now say a few words about these rules:

- LOCAL, COM, COND-THEN, COND-ELSE need no explanation.
- Rule CALL-FIRST, CALL-ENTER deal with running a procedure. The motivation behind the seemingly complicated rules goes beyond the scope of this work. What needs to be noted is that the rule CALL-FIRST looks up the procedure definition from the context, performs *processes substitutions* to replace the *formal parameters* with the *arguments* and inserts it to the continuation of the running choreography. The rule CALL-ENTER removes one by one the *runtime terms* introduced by CALL-FIRST.
- Rule DELAY captures in choreographies the notion that processes are independent of each other.
- Rule DELAY-COND, models the concurrent execution of instructions that are independent of a conditional, thus complementing DELAY.

### 2.1.4 Notes on expressivity

The presented language can be shown to be Turing Complete [3]. For Turing-complete languages, any *nontrivial extensional* (i.e., semantic) property of programs is undecidable [15]. If a property depends only on the function computed by a program and holds for some but not all computable functions, then there is no algorithm that always decides whether an arbitrary program has that property. The property we will consider in the main contribution of this work and which will be introduced in the next chapter is semantic in this sense, so no complete decision procedure exists in general.

### 2.1.5 Notes on small-step semantics

Because big-step semantics collapses the entire execution into a single relation between initial and final states, it inherently lacks the granularity required to represent instruction-level reordering. Consequently, it is inapplicable to model out-of-order execution: a feature that fundamentally relies on the scheduling and interleaving of micro-steps.

## 2.2 Information flow analysis

In modern computing systems, the handling and protection of data is of critical importance. [7]. With the proliferation of interconnected systems, sensitive data such as personal information, financial records, and classified communications is constantly processed, transmitted, and stored [21]. Ensuring that this information is handled securely and does not unintentionally or maliciously flow to unauthorized entities is a major challenge in computer science and software engineering [21]. Information Flow Analysis [4] is a set of techniques aimed at analyzing how information propagates through a program or system, with the goal of identifying potential leaks or violations of security policies.

In Denning's formulation [4], secure information flow means that all data transfers must conform to a *flow policy* defined by a relation $\rightarrow$, where $A \rightarrow B$ indicates that information is permitted to flow from *security class* $A$ to *security class* $B$. *Security classes* correspond to disjoint classes of information. They are intended to encompass *security classifications*. Each object in the system is bound to a security class.

Information flows can arise in two principal ways:

- **Explicit flows** occur when operations like assignment or message passing directly transfer information from one location to another.
- **Implicit flows** occur when the control structure of the program (e.g., conditionals or loops) induces a dependency between variables, such that the value of one variable may be inferred from the control decisions influenced by another, without any explicit data transfer. For example:

```
public = 1
if secret == 0:
    public = 0
```

Listing 2.1: Example of implicit flow

This code creates an implicit flow from `secret` to `public`, even though `public` is not explicitly assigned from `secret`.

Secure flow analysis of any system must capture both types of flows to ensure that all data transfers respect the flow relation.

A central result of Denning's work is the recognition of a *lattice structure* over the flow relation. The lattice ensures that:

- Every pair of classes has a unique *least upper bound* (join $\sqcup$) and *greatest lower bound* (meet $\sqcap$). If a value computed from multiple sources is assigned to a target, then the *composite class* of the sources (computed using the *least upper bound* operator) must be allowed to flow into the class of the target.
- Security of individual operations implies the security of sequences of operations, by transitivity of $\rightarrow$

Information Flow Analysis can be conducted using:

- **Static analysis**, which inspects code without executing it to verify that all potential flows are secure.
- **Dynamic analysis**, which tracks actual flows during execution by tagging and monitoring data.
- **Hybrid approaches**, which use static guarantees and insert runtime checks where necessary.

Denning's *Information Flow Analysis* provides a formal framework for reasoning about how data propagates through programs, enabling the development of tools and techniques that can be evaluated against a mathematically grounded, lattice-based policy structure.

**Is information-flow control enough?** Information-flow analysis, as usually formulated at the language level, reasons about flows that are explicit in values and implicit in control flow according to the operational semantics [16]. *Side channels* (e.g. timing, termination, resource usage, cache effects, message sizes, or scheduler-dependent behavior) fall outside this view unless the semantics and the attacker observation model explicitly make them observable. Any security guarantee should therefore be read *relative to the chosen observation model*. When side channels matter, they can be brought into scope by enriching the semantics with cost or timing observables and adopting timing-/step-sensitive definitions [1].

## 2.3 Non-Interference

Denning's work [4] is primarily concerned with the design and specification of information flow policies rather than their enforcement in concrete programming languages. Notably, the lattice model does not define how to formally relate a program's

execution semantics to the flow policy. While the model is sound as a representation of policy, it operates at an abstract level, and leaves open the question of how to *rigorously* ensure that actual programs respect the intended information flow restrictions.[2]

The notion of non-interference [5] provides a semantic formalization that addresses this limitation. Informally, non-interference requires that variations in high-security (confidential) inputs must not influence low-security (observable) outputs [17]. This condition captures the intuitive idea that secret data should not interfere with what an external observer can learn from the behavior of a program. Importantly, non-interference can be defined with respect to the program's operational semantics, thereby allowing for formal soundness proofs of enforcement mechanisms that guarantee compliance with the security policy [17].

In contrast, purely dynamic enforcement mechanisms, such as runtime monitors, are unable to detect certain classes of implicit information leaks [16]. Let us look back at the example code in 2.1. Dynamic mechanisms typically monitor only the path that is actually taken during execution. If this program is executed with `secret` $\neq 0$, the conditional branch is skipped and no assignment to `public` occurs. A dynamic monitor observing this trace would see no operation involving secret, and thus incorrectly conclude that no illegal information flow has occurred[3]: an attacker observing the final value of public still gains information: `public` having value 1 implies that the condition `secret == 0` did not hold. That is, the attacker can rule out one possible value for secret. While the leaked information may appear small, it is nonetheless a violation of confidentiality.

This illustrates a fundamental shortcoming: dynamic enforcement cannot reason about *potential* flows along untaken branches. Since information flow security is a property of all possible executions [16] [4], such mechanisms are inherently incomplete in capturing the full security implications of a program. In contrast, static approaches can be equipped to reason about all program paths [17] and thus offer a more precise and rigorous framework for enforcing confidentiality.

### 2.3.1 Definition

Let us consider a simple imperative programming language [19] with commands such as assignments, sequencing, conditionals, and loops. A program state $s$ is typically modeled as a mapping from variables to values [19], that we will partition into *high* and *low* components: $s = \langle s^h, s^l \rangle$, where $s^h$ contains high-security data and $s^l$ contains low-security data.

[2] In the concluding section of her paper, Denning briefly surveys various enforcement mechanisms, including compiler-based techniques and hardware support. However, this survey is based on intuitive arguments rather than being a formal account.

[3] unless monotonically increasing *label creep* [16] is accepted as result of the analysis

The semantics of a program $C$ is given [8] [13] by a function $[\![C]\!] : S \to S_\perp$, where $S$ is the set of program states and $S_\perp = S \cup \{\perp\}$ includes a special element $\perp$ representing non-termination.

Let $s_1 \equiv_L s_2$ denote that two states are *low-equivalent* i.e. they agree on all low-security variables: $s_1^l = s_2^l$.

Then, the formal definition of non-interference is [17] [5]:

$$\forall s_1, s_2 \in S.\ s_1 \equiv_L s_2 \Rightarrow [\![C]\!](s_1) \approx_L [\![C]\!](s_2)$$

Here, $\approx_L$ denotes *observational equivalence*[4] from the perspective of a low-security observer. In a termination-sensitive setting [6], this relation is defined as follows:

$$[\![C]\!](s_1) \approx_L [\![C]\!](s_2) \quad \text{iff} \quad \begin{cases} [\![C]\!](s_1) = \perp \text{ and } [\![C]\!](s_2) = \perp, \\ \text{or} \\ [\![C]\!](s_1), [\![C]\!](s_2) \in S \text{ and } [\![C]\!](s_1) \equiv_L [\![C]\!](s_2), \end{cases} \tag{2.1}$$

This definition ensures that, for any two initial states that agree on low-security data, their respective executions are indistinguishable to an attacker who observes only low-security outputs and can detect (non-)termination.

### Termination Sensitivity

When formalizing non-interference, a key consideration is whether termination behavior should be treated as an observable effect [6]. This leads to two distinct variants of the property: *termination-sensitive non-interference* and *termination-insensitive non-interference*.

**Termination-sensitive non-interference** requires that secret inputs cannot affect *either* the final low-observable state *or* whether the program terminates [18]. Formally, in this setting the observational equivalence relation $\approx_L$ is defined as shown in 2.1. To illustrate why termination-sensitive non-interference may be preferable, consider the following program:

```
if secret == 0:
    while True: pass
```

[4]This notion of observational equivalence can be naturally extended to account for additional observables beyond final low-security state and termination behavior [16]. For instance, one may define $\approx_L$ to reflect distinctions based on execution time (capturing timing channels), on the sequence of outputs to public channels (capturing event traces), or on probabilistic distributions over outputs (capturing probabilistic leakage).

In this example, the secret variable influences whether the program terminates. Specifically, if `secret` is zero, the program diverges, otherwise it terminates immediately. Thus, an attacker who observes termination behavior can directly infer the value of `secret`, revealing confidential information through the program's termination. This scenario provides a strong rationale for adopting termination-sensitive non-interference in settings where termination or responsiveness is observable.

**Termination-insensitive non-interference** by contrast, assumes that non-termination is *not* observable by the attacker [5]. Under this weaker definition, the observational equivalence relation $\approx_L$ only requires that whenever two executions terminate, they yield indistinguishable low-observable states [17]:

$$[\![C]\!](s_1) \approx_L [\![C]\!](s_2) \quad \text{iff} \quad [\![C]\!](s_1), [\![C]\!](s_2) \in S \implies [\![C]\!](s_1) \equiv_L [\![C]\!](s_2)$$

In this setting, divergences influenced by secret data are allowed.

The choice between these two definitions ultimately depends on the attacker model assumed. Termination-sensitive non-interference provides stronger guarantees and is well-suited for high-assurance scenarios in which termination behavior is observable by the attacker. Termination-insensitive non-interference is weaker but simplifies analysis and enforcement by eliminating the need to handle issues related to termination or infinite loops, which is particularly important because non-interference cannot be enforced in a sound and precise manner in the presence of these behaviors [12].

### Non-Determinism

**TODO: parla delle difficoltà del parlare di programmi non-deterministici, e parla delle soluzioni adottate nella letteratura TODO: (BONUS: qui potrei parlare di decentralize label model, ma alla fine l'abbiamo abbandonato quindi ha senso solo se ne voglio poi parlare nelle future estensioni)**

## 2.4 Proof Assistants

**TODO: questa da fare meglio, più avanti** In our setting, proofs are not merely pen-and-paper artefacts but machine-checked derivations. Using a proof assistant (Lean) brings two benefits: (i) we precisely define the language, the type-system, and the semantics, closing gaps that often hide in informal presentations; and (ii) via Curry–Howard, the typing rules become executable code, yielding a verified typechecker that we can run on examples.

Proof assistants are software systems designed to aid in the construction and verification of formal proofs within a logical framework. Their foundations lie primarily on the Curry-Howard correspondence to prove soundness of the approach.

### 2.4.1 Calculus of Inductive Construction

The Calculus of Inductive Constructions (CIC) is a formal system that serves as the theoretical foundation for several modern proof assistants, most notably Coq and Lean. It combines two major features of type theory: dependent types and inductive types. Building upon the Curry-Howard correspondence, CIC enables a unified framework in which logical propositions are treated as types and proofs as programs. Unlike simpler systems based on the simply-typed lambda calculus, CIC supports abstraction over both terms and types, as well as the definition of data structures and inductive reasoning.

#### Dependent Types

The Calculus of Constructions (CoC) extends the simply-typed lambda calculus by introducing dependent function types. These allow the type of a function's output to depend on the value of its input. CoC distinguishes between three categories of entities:

- Terms which represent programs or proofs
- Types which classify terms, and
- Sorts which classify types (e.g., Type : $\text{Type}_1$ or Prop : Type in Lean).

A key construct is the dependent product type, written $\Pi(x : A), B(x)$, which generalizes both universal quantification and function types. For instance:

$\Pi(A : \mathit{Type}), A \to A$ This type represents the polymorphic identity function. Under Curry–Howard, this also corresponds to the proposition: "for all A, A $\to$ A," and a term of this type is a proof of that proposition.

The CoC, however, lacks mechanisms for defining user-specified data types such as natural numbers or lists. This gap is addressed in CIC by the addition of inductive types.

### 2.4.2 Lean theorem prover

Lean is an open-source interactive theorem prover and programming language, developed by Microsoft Research and Carnegie Mellon University. At its heart is a lightweight trusted kernel built on dependant type theory, which can be configured to work using the CIC logical framework.

### Kernel

### Tactics framework

Lean provides tactics as an approach to constructing proof terms. We can view a term as a representation of a construction or mathematical proof; tactics are commands, or instructions, that describe how to build such a term.

### White box automation

### Mathlib3

# Chapter 3

# Enforcement of the Non-Interference property in Choreographies

The main contribution of this thesis is the development of a mechanism to check the compliance of a choreography against an user specified flow policy. As argued previously, it is advantageous to develop this system statically. A static type-system, defined as a type judgment relation, is the natural vehicle to enforce non-interference in choreographies because it turns a semantic security requirement into a syntactic discipline that can be checked algorithmically, integrated into compilation, and compositional on the inductive structure of the program. Volpano, Smith, and Irvine [17] established the standard soundness connection between such typing judgments and non-interference, providing a proof-theoretic route to a semantic guarantee. In this context *soundness* is defined as follows: A type-system is considered sound if for any flow policy $\Pi$, every program that is well-typed under $\Pi$ is *semantically compliant* with $\Pi$; that is, it satisfies termination insensitive non-interference.

The following work is greatly inspired by previous standard techniques for defining and proving soundness of type judgments [11] [20], applied to the case of the Choreographic Language defined in the previous chapter.

**Roadmap.** The remainder of this chapter is organized as follows: Section 3.1 formalizes the flow policy as a security lattice; Section 3.2 develops the main intuitions that motivate the type system; and Section **??** presents the formal definition of the type system.

## 3.1 Definition of the flow-policy

*Security labels* are elements of a complete lattice $(\mathscr{L}, \sqsubseteq)$ endowed with a bottom element $\bot$ s.t. $\forall l \in \mathscr{L}, \bot \sqsubseteq l$. These labels capture Denning's notion of security classes, where every object manipulated by the program has an associated security class.

A *process security labelling* $\gamma$ models the security class *i.e. security label* associated with the variables accessed by a process. Formally, a process store is a function from variables to security labels:

$$\gamma : \mathbf{Var} \longrightarrow \mathscr{L}$$

We write **SecPLab** for the set of all process security labellings. Similarly to what we defined for *choreographic stores* we define a *choreographic security labelling* $\Gamma$ as a map from process names to their respective process labelling. Formally,

$$\Gamma : \mathbf{PName} \longrightarrow \mathbf{SecPLab}.$$

We write **SecCLab** for the set of all choreographic security labellings.

In the non-interference framework, the flow-policy is fully specified by the security labelling: we forbid any flow of information from an object (in this case, variable) with an *higher* security associated label towards an object with a *lower* associated label.[1]

## 3.2 Intuitive presentation of the type-system

### 3.2.1 A motivating example for Program Counter Labelling

As we saw, to ensure non-interference we need to consider both *implicit* and *explicit* flows. Let us see some examples and build the intuition behind the type-system:

- **explicit flows**: Let us try to build a type-system *only* concerned with verifying explicit flows, trying to verify the following choreography:

  $$p.x := y + z;\ p.x \to q.x;\ \mathbf{0}$$

  The system can be built by composing constraints on the security labels of the variables, more precisely it is easy to see how this program would follow the flow-policy $\Gamma$ if:

  $$\begin{aligned} \Gamma\ p.y\ \sqcup\ \Gamma\ p.z &\sqsubseteq \Gamma\ p.x \\ \Gamma\ p.y &\sqsubseteq \Gamma\ q.x \end{aligned}$$

[1] The notion of higher, lower are defined naturally from the partial order relation $\sqsubseteq$

- **implicit flows**: Now, building from the previous example, let us introduce an implicit flow of information:

$$\text{if } p.(a == 0) \text{ then } p.x := 0;\ \mathbf{0} \text{ else } p.x := y + z;\ \mathbf{0};\ p.x \to q.x;\ \mathbf{0}$$

  We need to consider a further element: *the security label of the execution context*. In this particular case, the security label of the context depends on the security label of $p.a$ for the assignment instruction. We can thus update our to constraints to:

$$\Gamma\ p.y\ \sqcup\ \Gamma\ p.z\ \sqcup\ \Gamma\ p.a \sqsubseteq \Gamma\ p.x$$
$$\Gamma\ p.y \sqsubseteq \Gamma\ q.x$$

  We model this by keeping track of $pc \in \mathscr{L}$ in the *assumption* of the type judgment.

### 3.2.2 Typing procedure calls

Most of the foundational work on non-interference [16] builds type-systems for a *while language*, while find ourself having to develop one for a language supporting recursive procedures.

For this goal, we introduce a *procedure security context* **SecFunCtx**

$$\Delta : \mathbf{ProcName} \times \mathscr{L} \to 2^{\mathbf{SecCLab}}$$

such that, for every $X \in \mathbf{ProcName}$, $pc \in \mathscr{L}$, $\Gamma\ \in\ \Delta\ X\ pc$, then the body of $X$ is *well typed* under $\Gamma$ and $pc$.

Further discussion will follow on how to compute the context which carries this property, as of now its existence will simply be assumed.

### 3.2.3 Putting it all together

We are now ready to define our type judgment relation:

- Local expressions are assigned a security label by taking the supremum of the security labels of the occurring variables
- Instructions that modify the store (assign, send) use the check discussed previously (considering also the value of $pc$) so explicit and implicit flows are handled uniformly.
- Conditionals lift $pc$ with the guard's security label on both branches, preventing leaks through control flow.

- Calls are verified against $\Delta$ which lets us reason about recursion without unrolling.
- Choreographies compose by conjunction: sequencing preserves well-typing if each component does.

## 3.3 Formal definition of the Type System

### 3.3.1 Judgment relation

We use three typing judgments, one for every syntactic category used to define Choreographies. We will overload the $\vdash$ symbol. The relation used will be clear by the context. The three relations are denoted as follows:

- **Expressions** $\Gamma\ p \vdash e : \ell$
- **Instructions** $\Delta; \Gamma; pc \vdash i$
- **Choreographies** $\Delta; \Gamma; pc \vdash C$

Where $l \in \mathscr{L}$

### 3.3.2 Typing Rules for Expressions

Expressions are always considered local and every local function is considered *deterministic* and *total*. Thus we define

$$\cdot \vdash \cdot : \cdot \quad : \quad \mathbf{SecPLab} \to \mathbf{Expr} \to \mathscr{L}$$

As the smallest relation following the following inference schema:

**Constant**

$$\Gamma\ p \vdash c : \bot$$

**Variable**

$$\Gamma\ p \vdash x : \Gamma\ p\ x$$

**N-ary function**

$$\frac{\Gamma\ p \vdash e_1 : \ell_1 \quad \cdots \quad \Gamma\ p \vdash e_n : \ell_n \qquad \ell' = \sqcup_{i=1}^{n} \ell_i}{\Gamma\ p \vdash f(e_1, \ldots, e_n) : \ell'}$$

We assume primitive functions to be label-preserving and do not introduce any extra leak of information. The typing rule thus assumes that functions do not introduce additional sensitivity; the result is at least as sensitive as the argument.

### 3.3.3 Typing Rules for Instructions

**Assignment**

$$\frac{\Gamma\ p \vdash e : \ell' \quad \ell' \sqcup pc \sqsubseteq \Gamma\ p\ x}{\Delta; \Gamma; pc \vdash\ p.x := e}$$

**Communication**

$$\frac{\Gamma\ p \vdash e : \ell' \quad \ell' \sqcup pc \sqsubseteq \Gamma\ q\ x}{\Delta; \Gamma; pc \vdash\ p.e \to q.x}$$

We can see how communication is treated as an assignment between processes. This requires as assumption that *communication channels are private*, i.e. no other party outside of sender and receiver can read the content of the channel **TODO: sta parte è interessante espanderla in possibili estensioni**

**Selection and Runtime Call Term**

$$\Delta; \Gamma; pc \vdash\ p \to q[\mathsf{L}] \quad \text{and} \quad \Delta; \Gamma; pc \vdash\ X(\vec{p}\,).C$$

Both terms are administrative, carry no data, and do not influence information-flow. We consider them as always well-typed.

**Conditionals**

$$\frac{\Gamma\ p \vdash e : \ell' \quad \Delta; \Gamma;\ \ell' \sqcup pc \vdash C_1 \quad \Delta; \Gamma;\ \ell' \sqcup pc \vdash C_2}{\Delta; \Gamma; pc \vdash\ \textbf{if}\ p.e\ \textbf{then}\ C_1\ \textbf{else}\ C_2}$$

**Procedure Calls**

$$\frac{\Gamma' \in \Delta(X, pc) \quad \Gamma[\vec{q} \mapsto \vec{p}] \equiv_{\{\vec{q}\}} \Gamma'}{\Delta; \Gamma; pc \vdash\ X(\vec{p})}$$

Where $\vec{q}$ is the list of formal parameters of the procedure $X$ in the context, and $\vec{p}$ is the list of arguments applied to the procedure call.
Let us unpack the meaning of this rule: As we know from the definition of $\Delta$, $\Gamma'$ will be a security context that well-types the body of $X$. Let us now focus on the second antecedent of the rule by defining the two new operators:

**Context renaming** Given lists of processes $\vec{q} = q_1, \ldots, q_n$ and $\vec{p} = p_1, \ldots, p_n$ (always considered of equal length), we write $\Gamma[\vec{q} \mapsto \vec{p}]$ for the environment obtained from $\Gamma$ by updating the context pointed by each $p_i$ to mirror the one pointed by $q_i$. Formally (in the scalar case):

$$\Gamma[q/p]\ r \triangleq \begin{cases} \Gamma\ p & \text{if } q = r \\ \Gamma\ r & \text{otherwise} \end{cases}$$

**TODO: ad un certo punto dovrai dire qualcosa sul fatto che l'estensione "vettoriale" ha associatività al contrario**

**Restricted equality** For a finite set of processes $S$, write $\Gamma \equiv_S \Gamma'$ when $\Gamma$ and $\Gamma'$ agree on all variables of all processes in $S$. Formally, considering extensional equality between maps:

$$\Gamma \equiv_S \Gamma' \quad := \quad \forall r \in S, \quad \Gamma\ r = \Gamma'\ r$$

Given these two definitions, we can explain the meaning of the second antecedent as follows: For every process $q_i$ in the formal parameters then $\Gamma'\ q_i$ is the same as $\Gamma\ p_i$, with $p_i$ as the process in the arguments corresponding to $q_i$. [2]

### 3.3.4 Typing Rules for Choreographies

The type-system composes on the instructions making up a choreography:

**Sequencing**

$$\frac{\Delta; \Gamma; pc \vdash i \quad \Delta; \Gamma; pc \vdash C}{\Delta; \Gamma; pc \vdash\ i\ ;\ C}$$

**Empty Choreography**

$$\Delta; \Gamma; pc \vdash\ \mathbf{0}$$

[2]The rule could be less strict: we could only consider single variables restriction instead of process-level equality. The choice of using process equality was made to not over-complicate the definitions and the proof to follow

# Chapter 4

# Soundness of the Type System

For the type-system to be interesting we need to prove it's soundness with respect to termination insensitive non-interference against the reference semantics.

## 4.1 Overall statement and proof obligations

Fix a public observation level $low \in \mathscr{L}$. Recall that $[\![C]\!] : S \to S_\perp$ is the (partial) denotational semantics of choreographies into states $S$ extended with $\perp$ (non-termination), that $s_1 \equiv^{\Gamma}_{low} s_2$ means the two states agree on all variables labelled $\sqsubseteq low$ in $\Gamma$.

The soundness theorem states that *well-typed programs satisfy TINI*:

$$\Delta; \Gamma; \perp \vdash C \quad \Longrightarrow \quad \forall s_1, s_2 \in S.\ s_1 \equiv^{\Gamma}_{low} s_2 \ \Rightarrow\ [\![C]\!](s_1) \approx_? [\![C]\!](s_2).$$

To fully specify this theorem, we need to define an equivalence relation that encodes the concept of *low-equivance under TINI*. To join this need with the semantics presented in 2.1 we introduce a *natural semantics* [8] for Choreographies. We define a relation:

$$\langle C, \Sigma, \mathscr{C} \rangle \Downarrow^M \Sigma'$$

Where:

- $C \in$ **Chor**: Choreography
- $\Sigma, \Sigma' \in S$: Choreographic store
- $\mathscr{C}$: Procedure context
- $M \in$ **List TLabel**: sequence of zero or more **TLabel**. We will use list notation standard to functional programming languages.

defined as the smallest relation following the following schema:

$$\langle \mathbf{0}, \Sigma, \mathscr{C} \rangle \Downarrow^{[]} \Sigma \qquad \frac{\langle C, \Sigma, \mathscr{C} \rangle \xrightarrow{\mu} \langle C', \Sigma', \mathscr{C} \rangle \qquad \langle C', \Sigma', \mathscr{C} \rangle \Downarrow^{M} \Sigma''}{\langle C, \Sigma, \mathscr{C} \rangle \Downarrow^{\mu::M} \Sigma''}$$

We can thus now state the *termination insensitive non-interference theorem* as follows:

$$\begin{aligned} & \Delta; \Gamma; \bot \vdash C \Rightarrow \Sigma_1 \equiv^{\Gamma}_{low} \Sigma_2 \\ \Rightarrow & \langle C, \Sigma_1, \mathscr{C} \rangle \Downarrow^{M_1} \Sigma_1' \Rightarrow \ \langle C, \Sigma_2, \mathscr{C} \rangle \Downarrow^{M_2} \Sigma_2' \\ \Rightarrow & \Sigma_1' \equiv^{\Gamma}_{low} \Sigma_2' \end{aligned} \tag{4.1}$$

To justify this statement we rely on a small set of proof obligations that connect typing, expression evaluation, and the operational/denotational semantics. Each obligation is stated formally and followed by a short explanation of the intuition behind it.

### Properties of procedure context

The main intuition behind the introduction of $\Delta$ for typing procedure calls was explained previously.

**Context subsumption** We say that $\Delta$ satisfies *context subsumption* if lowering the control level always maintains typing: for every procedure name $X$, every pair of program-counter labels $pc, pc' \in \mathscr{L}$, and every security environment $\Gamma$,

$$\Gamma \in \Delta\ X\ pc\ \wedge\ pc' \sqsubseteq pc\ \Longrightarrow\ \Gamma \in \Delta\ X\ pc'$$

Typing a call to $X$ that is valid under a more restrictive (higher) control $pc$ must remain valid when the analysis proves that control has become more public. This is used in the soundness proof whenever we *lower* the current program counter along a derivation (e.g., after leaving a high guard).

**Well-formed procedure context** We say that the procedure context $\mathscr{C}$ is *well formed* if every procedure lists all the participants that actually appear in its body. Formally, for every definition $X(\vec{p}) = C \in \mathscr{C}$,

$$\mathrm{pn}(C) \subseteq \{\vec{p}\}.$$

This property ensures that the interface $X(\vec{p})$ exposes exactly the processes that $C$ may mention, preventing references to undeclared processes and simplifying the typing of calls. In addition to this, this notion of *well-formedness* for the procedure context is a less stringent version of the notion given by Montesi's book [9]. The other properties needed for a context to be well-formed go beyond the scope of this thesis, thus we will omit them.

**Well-typed Security Procedure context** We require the typing context $\Delta$ to be consistent with the declaration context $\mathscr{C}$: every declared procedure must typecheck under every security environment that $\Delta$ admits for it (at any program counter). Formally, for every clause $X(\vec{p}) = C \in \mathscr{C}$ and every $pc \in \mathscr{L}$,

$$\Delta\ X\ pc = \mathcal{G} \implies \forall \Gamma \in \mathcal{G}.\ \Delta; \Gamma; pc \vdash C.$$

Here $\mathcal{G}$ is the set of admissible security environments for the body $C$ at program counter $pc$. It is allowed that, for some $X$ and $pc$, the lookup yields an empty set of admissible environments, i.e. $\Delta(X, pc) = \varnothing$. In that case no $\Gamma$ can satisfy the side condition $\Gamma \in \mathcal{G}$ in the typing rule for calls to $X$, so any call to $X$ at that $pc$ is untypable: the derivation stops at the $\Delta$-lookup premise. Intuitively, this expresses that $X$ cannot be used at program counter $pc$.

**Freshness of formal parameters.** We assume a Barendregt-style convention for procedure parameters [2]: in the procedure-definition context $\mathscr{C}$, all formal parameters are chosen *sufficiently fresh*. Concretely, for every clause $X(\vec{p}) = C \in \mathscr{C}$:

1. the names in $\{\vec{p}\}$ are pairwise distinct;
2. for every other choreography $C'$ s.t. $C \neq C'$, the parameters $\{\vec{p}\}$ do not clash with the process names that occur in $C'$:

$$\{\vec{p}\} \cap \mathrm{pn}(C') = \varnothing.$$

Intuitively, parameters bind the process names used inside $C$; by choosing them distinct and disjoint from the names mentioned elsewhere in $\mathscr{C}$, we avoid spurious name capture when instantiating $X$ with actual participants (e.g. at call sites or during inlining). We will use $\alpha$-renaming implicitly to maintain this invariant throughout.

### Properties of expressions

The semantics referenced [9] is mostly underspecified when dealing with local expressions, with no indication on how to evaluate them in the general case. We wish to keep as close as possible to the reference, thus we abstain from specifying a concrete semantics. We will limit ourselves to constraining local evaluation to being *deterministic* Formally: given a process store $\sigma$, and expression $e$ and two values $v_1, v_2$:

$$\sigma \vdash e \downarrow v_1 \Rightarrow \sigma \vdash e \downarrow v_2 \Rightarrow v_1 = v_2$$

We assume this to prevent guards and right-hand sides from introducing spurious nondeterminism that a low observer could notice.

## 4.2 Instrumented Choreographies

Unfortunally we are not able to do the proof directly. We want to be able to use induction on the lenght of the computation, but given the small-step nature of the semantics, we need to *carry over* some information from previous execution states. More concretely: given a flow policy, the same configuration state could be non-interferent or not depending on previous states in the computation. Let us see an example:

$$\text{if } p.(a == 0) \text{ then } p.x := 0;\ \mathbf{0} \text{ else } p.x := y + z;\ \mathbf{0};\ \mathbf{0}$$

We know that this choreography follows non-interference for some flow-policy. Let us denote the choreography with $c$. Given a $\mathscr{C}$ and $\Sigma_1, \Sigma_2$ such that $\Sigma_1\ p.a = 0, \Sigma_2\ p.a = 1$ then we have the following transitions:

$$\langle c, \Sigma_1, \mathscr{C}\rangle \xrightarrow{\tau@p} \langle p.x := 0;\ \mathbf{0}, \Sigma_1, \mathscr{C}\rangle$$
$$\langle c, \Sigma_2, \mathscr{C}\rangle \xrightarrow{\tau@p} \langle p.x := y + z;\ \mathbf{0}, \Sigma_2, \mathscr{C}\rangle$$

Clearly,

$$p.x := 0;\ \mathbf{0} \neq p.x := y + z;\ \mathbf{0}$$

thus the induction hypothesis could not be used. To solve this, we need a way to be able to differentiate when choreogreographies are able to be different and when they are not.

We do this by introducing *brackets* around code able to differ.

**intuition** Given a fixed element *low* in $\mathscr{L}$, we consider anything inside bracket as being *not observable* by a partecipant of level $l \sqsubseteq$ *low*. We also call denote the partecipant as a *low-observer*.

### 4.2.1 Syntax

We instrument choreographies with *brackets* to mark code fragments that are allowed to differ across alternative executions. The syntax of (instrumented) choreographies is given below; expressions are left abstract.

| | |
|---|---|
| $p, q \in \mathsf{Pid}$ | (process names) |
| $x \in \mathsf{Var}$ | (local variables) |
| $X \in \mathsf{ProcName}$ | (procedure names) |
| $L \in \mathsf{Label}$ | (selection labels) |
| $e \in \mathsf{Expr}$ | (expressions) |

Choreographies:

$$
\begin{array}{rcll}
C \in [\mathsf{Chor}] & ::= & \mathbf{0} & \text{(termination)} \\
& | & C_1 \,;\, C_2 & \text{(sequencing)} \\
& | & [C] & \text{(bracketed fragment)} \\
& | & p.x := e & \text{(assignment)} \\
& | & p.e \;\rightarrow\; q.x & \text{(communication)} \\
& | & \textbf{if } p.e \textbf{ then } C_1 \textbf{ else } C_2 & \text{(conditional at } p\text{)} \\
& | & X\langle \vec{p} \rangle & \text{(procedure call with participants)}
\end{array}
$$

We note that [Chor] differs from Chor on a few key points:

- *Flattening of instructions and choreographies:* We want to be able to put inside the brackets a sequence of instructions of arbitrary lenght (possibily zero), thus we want to be able to differentiate $[\mathbf{0}]$ and $\mathbf{0}$.
- *Removal of the runtime term and label selection instructions:* This was not necessary but simplifies the soundness proof. It's justification is that both instructions have no impact on the final computed stores and are needed by choreographies to deal with projection [9] which goes beyond the scope of this thesis.

### 4.2.2 Lowering and lifting of [Chor]

We introduce an operator $\lfloor \cdot \rfloor : \mathbf{[Chor]} \to \mathbf{Chor}$ that *removes* the instrumentation from a choreography. The operator is defined by structural recursion on the structure of $C$:

$$
\begin{aligned}
\lfloor \mathbf{0} \rfloor &\triangleq \mathbf{0} \\
\lfloor C_1 \,;\, C_2 \rfloor &\triangleq \lfloor C_1 \rfloor \fatsemi \lfloor C_2 \rfloor \\
\lfloor [C] \rfloor &\triangleq \lfloor C \rfloor \\
\lfloor p.x := e \rfloor &\triangleq p.x := e; \mathbf{0} \\
\lfloor p.e \;\rightarrow\; q.x \rfloor &\triangleq p.e \;\rightarrow\; q.x; \mathbf{0} \\
\lfloor \textbf{if } p.e \textbf{ then } C_1 \textbf{ else } C_2 \rfloor &\triangleq \textbf{if } p.e \textbf{ then } \lfloor C_1 \rfloor \textbf{ else } \lfloor C_2 \rfloor; \mathbf{0} \\
\lfloor X(\vec{p}) \rfloor &\triangleq X(\vec{p}); \mathbf{0}
\end{aligned}
$$

We also define an operator $\lceil \cdot \rceil$ that turns a **Chor** into the corresponding **[Chor]** by copying it's shape.

We note that not every **Chor** is in the domain of $\lceil \cdot \rceil$. As a demonstrating example, let's consider:

$$\lceil p \rightarrow q[L]; \mathbf{0} \rceil$$

We would not know how to create an equivalent term in **[Chor]**. This problem will be properly addressed in 4.3.1. For now we will just define $\lceil \cdot \rceil$ as being *the inverse of lowering*, well defined only for choreographies such that

$$C = \lfloor \lceil C \rceil \rfloor \tag{4.2}$$

Both lowering and lifting are defined also on *procedure context*, as follows:

$$\lfloor \mathscr{C} \rfloor \triangleq \{X(\vec{p}) = \lfloor C \rfloor \mid X(\vec{p}) = C \in \mathscr{C}\}$$
$$\lceil \mathscr{C} \rceil \triangleq \{X(\vec{p}) = \lceil C \rceil \mid X(\vec{p}) = C \in \mathscr{C}\}$$

### 4.2.3 Low equivalence of [Chor]

We write $C_1 \approx_{\mathsf{low}} C_2$ to denote *low–equivalence* between choreographies. Intuitively, $\approx_{\mathsf{low}}$ compares choreographies structurally, but *forgets the contents of bracketed fragments*: any two bracketed subterms are considered equivalent, independently of what they contain.
Formally, $\approx_{\mathsf{low}}$ is the *least* relation on [Chor] closed under the following rules:

$$\frac{C_1 \approx_{\mathsf{low}} C_2 \quad C_1' \approx_{\mathsf{low}} C_2'}{C_1 \,;\, C_1' \;\approx_{\mathsf{low}}\; C_2 \,;\, C_2'} \qquad [C_1] \;\approx_{\mathsf{low}}\; [C_2]$$

$$\frac{p = p' \quad e = e' \quad C_{11} \approx_{\mathsf{low}} C_{21} \quad C_{12} \approx_{\mathsf{low}} C_{22}}{\textbf{if } p.e \textbf{ then } C_{11} \textbf{ else } C_{12} \;\approx_{\mathsf{low}}\; \textbf{if } p'.e' \textbf{ then } C_{21} \textbf{ else } C_{22}}$$

For all the remaining constructors, low–equivalence coincides with syntactic equality (shape and parameters must match). Concretely:

$$\mathbf{0} \;\approx_{\mathsf{low}}\; \mathbf{0}, \qquad p.x := e \;\approx_{\mathsf{low}}\; p.x := e, \qquad p.e \to q.x \;\approx_{\mathsf{low}}\; p.e \to q.x,$$

$$X\langle \vec{p} \rangle \;\approx_{\mathsf{low}}\; X\langle \vec{p} \rangle.$$

By construction, $\approx_{\mathsf{low}}$ is an equivalence relation and a congruence for sequencing and conditionals; its only non-syntactic identification is the equation of bracketed fragments.

### 4.2.4 Well formedness of [CStore]

Before being able to present the instrumented semantics, we need to instrument the **CStore** to encode the same notion of *value not observable by a low-observer*.

We do this by extending the previous definitions, we introduce a difference between a *value* and a *high-value* (i.e. bracketed value).

$$\begin{aligned} [\mathsf{CStore}] &: \mathsf{PName} \to [\mathsf{PStore}] \\ [\mathsf{PStore}] &: \mathsf{Var} \to [\mathsf{Val}] \\ [\mathsf{Val}] &: \mathsf{Val} \mid [\ \ \mathsf{Val}\ \ ] \end{aligned}$$

We then say that a **[CStore]** $[\Sigma]$ is *well formed* with respect to a $\Gamma$ (i.e. $\Gamma \vdash [\Sigma]$) when the following property is satisfied for all $p, x, v$:

$$[\Sigma]\ p\ x = [v] \iff \Gamma\ p\ x \not\sqsubseteq low$$

Thus encoding with $\sqsubseteq$ the notion of *observability*: $a \not\sqsubseteq low$ means that *low* can not *see* the value of $a$

The extension of *well-formedness* to **[PStore]** is natural.

As we did for choreographies, we define a *lifting* function $\lceil\cdot\rceil^{\Gamma}$ for **CStore**. The main difference is that the lifting function will maintain well-formedness of the created **[CStore]** with respect to $\Gamma$. That is, it will choose bracketing of values depending on the security label in $\Gamma$ of the associated variables.

### 4.2.5 Correctness of [CStore]

We define a function $\lfloor\cdot\rfloor$ polimorfically on **[Val]**, **[PStore]** and **[CStore]**. This function *lowers* the bracketed object to it's reference counterpart. Formally, given an **[Val]** object $\mathbf{v}$

$$\mathbf{v} \mapsto \begin{cases} v & \text{if } \mathbf{v} = [v] \\ v & \text{if } \mathbf{v} = v \end{cases}$$

From now on every occurrance of $v$ has to considered as either **Val** or an **[Val]** object which is *not bracketed*. Every occurrance of $[v]$ is a *bracketed* **[Val]**. And we will use $\mathbf{v}$ when it is not specified either the object is bracketed or not.
We extend the *lowering* function to **PStore** and **CStore** by applying it to every stored value.

We say that a **[CStore]** $[\Sigma]$ is *correct* with respect to $\Sigma$ when

$$\lfloor[\Sigma]\rfloor = \Sigma$$

The extension of *correctedness* to **[PStore]** is natural.

### 4.2.6 Low equivalence on [CStore]

We define a notion of low-equivalence on [**Val**]: Given $\nu_1, \nu_2$ then

$$\nu_1 \approx_{low} \nu_2 \quad \text{iff} \quad \begin{cases} \nu_1 = v_1 \text{ and } \nu_2 = v_2 \text{ and } v_1 = v_2 \\ \text{or} \\ \nu_1 = [v_1] \text{ and } \nu_2 = [v_2] \end{cases} \tag{4.3}$$

We extend the previous notion of *low*-equivalence between stores to exploit the bracket notation. Formally given $[\Sigma_1], [\Sigma_2]$, then $[\Sigma_1] \approx_{low} [\Sigma_2]$ if, for every $p, x$, then

$$[\Sigma_1]\ p\ x \approx_{low} [\Sigma_2]\ p\ x$$

The extension of *low-equivalence* to [**PStore**] is natural.

### 4.2.7 Semantics

Fix an observation level $low \in \mathscr{L}$ and a choreographic security labelling $\Gamma$. We define the new small-step operational semantics as:

$$\langle C, [\Sigma], \mathscr{C} \rangle \to \langle C', [\Sigma'], \mathscr{C} \rangle$$

**Assignments**

$$\frac{[\Sigma]\ p \vdash e \downarrow v \qquad \Gamma\, p\, x \not\sqsubseteq low}{\langle p.x := e,\ [\Sigma],\ \mathscr{C} \rangle \to \langle \mathbf{0},\ [\Sigma][p.x \mapsto [v]],\ \mathscr{C} \rangle}$$

$$\frac{[\Sigma]\ p \vdash e \downarrow [v]}{\langle p.x := e,\ [\Sigma],\ \mathscr{C} \rangle \to \langle \mathbf{0},\ [\Sigma][p.x \mapsto [v]],\ \mathscr{C} \rangle}$$

$$\frac{[\Sigma]\ p \vdash e \downarrow v \qquad \Gamma\, p\, x \sqsubseteq low}{\langle p.x := e,\ [\Sigma],\ \mathscr{C} \rangle \to \langle \mathbf{0},\ [\Sigma][p.x \mapsto v],\ \mathscr{C} \rangle}$$

**Communications**

$$\frac{[\Sigma]\ p \vdash e \downarrow v \qquad \Gamma\, q\, x \not\sqsubseteq low}{\langle p.e \to q.x,\ [\Sigma],\ \mathscr{C} \rangle \to \langle \mathbf{0},\ [\Sigma][q.x \mapsto [v]],\ \mathscr{C} \rangle}$$

$$\frac{[\Sigma]\ p \vdash e \downarrow [v]}{\langle p.e \to q.x,\ [\Sigma],\ \mathscr{C} \rangle \to \langle \mathbf{0},\ [\Sigma][q.x \mapsto [v]],\ \mathscr{C} \rangle}$$

$$\frac{[\Sigma]\ p \vdash e \downarrow v \qquad \Gamma\, q\, x \sqsubseteq low}{\langle p.e \to q.x,\ [\Sigma],\ \mathscr{C} \rangle \to \langle \mathbf{0},\ [\Sigma][q.x \mapsto v],\ \mathscr{C} \rangle}$$

Intuitively, these instrumented semantics for assignment and communication are needed to preserve the well-formedness invariants of **CStore**. Let us look at the assignment rules, since the communication ones follow similarly:

- If the destination cell is not low-observable ($\Gamma\, p\, x \not\sqsubseteq \mathit{low}$), we always store a bracketed value: $[\Sigma][p.x \mapsto [v]]$. This preserves the invariant that $[\Sigma]\ p\ x$ is bracketed exactly when $\Gamma\, p\, x$ is not observable at $\mathit{low}$.
- If the expression already evaluates to a bracketed value $[v]$, we propagate the bracket on write. This never *unbrackets* high information and thus cannot violate well-formedness (by soundness with respect to explicit flow well-typed programs will not allow storing $[v]$ into a low-labeled location).
- If the destination is low-observable ($\Gamma\, p\, x \sqsubseteq \mathit{low}$) and the expression evaluates to an unbracketed $v$, we update with $v$ (no brackets). This prevents spurious brackets in low cells and maintains that only non-low-observable locations carry bracketed values.

**Conditionals**

$$\frac{[\Sigma]\ p \vdash e \downarrow [\mathit{true}]}{\langle \textbf{if } p.e \textbf{ then } C_1 \textbf{ else } C_2,\ [\Sigma],\ \mathscr{C}\rangle \to \langle [C_1],\ [\Sigma],\ \mathscr{C}\rangle}$$

$$\frac{[\Sigma]\ p \vdash e \downarrow \mathit{true}}{\langle \textbf{if } p.e \textbf{ then } C_1 \textbf{ else } C_2,\ [\Sigma],\ \mathscr{C}\rangle \to \langle C_1,\ [\Sigma],\ \mathscr{C}\rangle}$$

$$\frac{[\Sigma]\ p \vdash e \downarrow [v] \qquad v \neq \mathit{true}}{\langle \textbf{if } p.e \textbf{ then } C_1 \textbf{ else } C_2,\ [\Sigma],\ \mathscr{C}\rangle \to \langle [C_2],\ [\Sigma],\ \mathscr{C}\rangle}$$

$$\frac{[\Sigma]\ p \vdash e \downarrow v \qquad v \neq \mathit{true}}{\langle \textbf{if } p.e \textbf{ then } C_1 \textbf{ else } C_2,\ [\Sigma],\ \mathscr{C}\rangle \to \langle C_2,\ [\Sigma],\ \mathscr{C}\rangle}$$

Intuitively, the bracketed rules capture the dependency of control flow on information not observable at $\mathit{low}$. If the guard reduces under $[\Sigma]$ to a *bracketed* truth value, then the selected continuation is executed in bracketed form $[C_i]$: this explicitly records that the branch choice depends on data above $\mathit{low}$ and, via the assignment and communication rules, forces all subsequent effects to remain bracketed and thus invisible at $\mathit{low}$. Dually, when the guard evaluates to an *unbracketed* boolean, the choice is already determined at $\mathit{low}$, so the corresponding unbracketed continuation $C_i$ proceeds normally.

**High-step**

$$\frac{\langle C,\ [\Sigma],\ \mathscr{C}\rangle \to \langle C',\ [\Sigma'],\ \mathscr{C}\rangle}{\langle [C],\ [\Sigma],\ \mathscr{C}\rangle \to \langle [C'],\ [\Sigma'],\ \mathscr{C}\rangle} \qquad \frac{}{\langle [\mathbf{0}],\ [\Sigma],\ \mathscr{C}\rangle \to \langle \mathbf{0},\ [\Sigma],\ \mathscr{C}\rangle}$$

**Sequencing**

$$\frac{\langle C_1,\ [\Sigma],\ \mathscr{C}\rangle \to \langle C_1',\ [\Sigma'],\ \mathscr{C}\rangle}{\langle C_1\,;\,C_2,\ [\Sigma],\ \mathscr{C}\rangle \to \langle C_1'\,;\,C_2,\ [\Sigma'],\ \mathscr{C}\rangle} \qquad \frac{}{\langle \mathbf{0}\,;\,C,\ [\Sigma],\ \mathscr{C}\rangle \to \langle C,\ [\Sigma],\ \mathscr{C}\rangle}$$

**Procedure calls**

$$\frac{X(\vec{q}) = C \in \mathscr{C}}{\langle X(\vec{p}), [\Sigma],\ \mathscr{C}\rangle \to \langle C[\vec{q}/\vec{p}],\ [\Sigma],\ \mathscr{C}\rangle}$$

**Multi-step transitions** For notational ease, we define $\cdot \twoheadrightarrow \cdot$ as the transitive, reflexive closure of $\cdot \to \cdot$.

**Local expression evaluation** Expression evaluation is a natural extension to the one presented in the reference semantics. The only aspect which needs careful consideration is the extension to **[Val]** of $\vdash f(\vec{v}) \downarrow v'$. The extension *carries* brackets from the arguments of the function to it's returned values, that is: given a function

$$f : \mathbf{Val}^* \to \mathbf{Val}$$

we define a *lifted* $[f]$ s.t. for every $\vec{\nu}, \vec{v}$ s.t. $\lfloor \vec{\nu} \rfloor = \vec{v}$ then $\lfloor f(\vec{\nu}) \rfloor = f(\vec{v})$ and, if is exists $v_i, \nu_i$ s.t. $\nu_i \in \vec{\nu} \wedge \nu_i = v_i$ then, there exists $v$ s.t. $[f](\vec{\nu}) = v$.

This makes our evaluation relation *correct* with respect to the reference (proved by simple structural induction). In this context we define correctness as returning an object that, once lowered, it agrees with the reference implementation.
Formally:

$$\Sigma \vdash e \downarrow v \Leftrightarrow ([\Sigma] \vdash e \downarrow \nu \wedge \lfloor \nu \rfloor = v)$$

It is to be noted that this property, combined with assuming the reference evaluation as deterministic, makes local expression evaluation for the instrumented semantics deterministic as-well.

### 4.2.8 Extension of the type system

Most of the type system for **[Chor]** carries over from the one for **Chor**, with a new rule to type *bracketed choreographies*:

$$\frac{\Delta; \Gamma; \ell' \vdash\ C \quad pc \sqsubseteq \ell' \quad \ell' \not\sqsubseteq low}{\Delta; \Gamma; pc \vdash\ [C]}$$

This rule makes sure that the content of $C$ can be typed as high, ensured by the first and last premise.

The second premise enforces that the label attached to the difference, $l'$, *dominates* the ambient program counter. In this way all influences of the current control flow, the implicit flows tracked by *pc*, are subsumed by the bracket's label $l'$. **TODO: questo è word babble, in realtà credo non serva a nulla ma ormai è tardi per cambiarlo**

## 4.3 Auxilliary lemmas

To be able to prove the main non-interference theorem as defined in 4.1 we will firstly assume a few auxilliary lemmas. Since The majority of the proof approach is taken directly from [11], we will abstain from reproducing them in this work.

### 4.3.1 Completeness lemma

Completeness is stated as follows:
For each

$$\langle C, \Sigma, \mathscr{C} \rangle \Downarrow^M \Sigma'$$

There exists $[\Sigma]'$ such that:

$$\langle \lceil \wr C \wr \rceil, \lceil \Sigma \rceil^\Gamma, \mathscr{C} \rangle \Downarrow [\Sigma]' \wedge \lfloor [\Sigma]' \rfloor = \Sigma' \tag{4.4}$$

The operator $\wr \cdot \wr$ will transform $C$ into the subset of **Chor** for which $\lceil \cdot \rceil$ is well-defined. Part of the completeness proof will be showing that the operator has no influence on the natural semantics.

Since **Chor** and **[Chor]** have multiple small but significant differences, we proceed the completeness proof by composing consecutive steps:

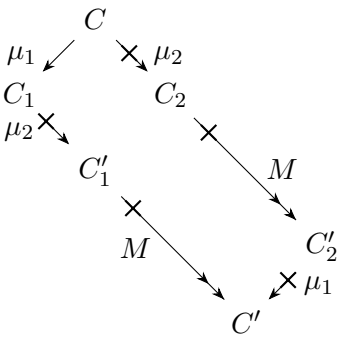
#### Removal of non-determinism in the semantics

We can easily verify how the semantics given in 2.1 is non-deterministic meaning that given the same configuration it admits multiple, different, execution steps. It is also to be noted that the instrumented semantics is defined as fully deterministic: any configuration that admits an execution step admits just one.

The first step to reach completeness is to prove that this non-determinism can be discarded. This goal is achieved by considering a semantics for choreography where the rules DELAY, DELAY-COND are omitted and where the rule CALL-FIRST has the following form:

$$\frac{X(\vec{q}\,) = C \in \mathscr{C} \qquad \vec{p} = p_1, \ldots, p_n}{\langle X(\vec{p}\,); C', \Sigma, \mathscr{C} \rangle \xrightarrow{\tau @ p_1} \langle p_2 : X(\vec{p}\,).C'; \ldots; p_n : X(\vec{p}\,).C'; C[\vec{q}/\vec{p}\,] \fatsemi C', \Sigma, \mathscr{C} \rangle} \text{ CALL-FIRST}$$

While a full proof of completeness of the deterministic semantics is not given, the following diagram gives the intuition: $C$ is able to do a step in the reference semantics and reach a $C_1$ configuration, while in the deterministic semantics (denoted with the crossed arrow) $C$ reaches a *different* $C_2$. Assuming the computation from $C$ as terminating in the reference semantics, it is always reachable a common configuration $C'$ from both $C_1$ and $C_2$ using the deterministic semantics.




The new semantics is deterministic as defined above: any configuration that admits an execution step admits just one, completely specified by the first instruction of the choreography. From now the deterministic semantics will be the semantics considered for **Chor**.

### Removal of irrelevant terms

We want to prove that terms not present in the image of $\lfloor\cdot\rfloor$ are not relevant to natural semantics to be able to use freely the $\lceil\cdot\rceil$ operator. We introduce the operator $\wr C \wr$ defined by structural recursion on $C$ as follows:

$$\begin{aligned}
\wr\ \mathbf{0}\ \wr &\triangleq \mathbf{0}\\
\wr\ p.e \to q.x;\ C'\ \wr &\triangleq p.e \to q.x;\ \wr C' \wr\\
\wr\ p \to q[L];\ C'\ \wr &\triangleq \wr C' \wr\\
\wr\ p.x := e;\ C'\ \wr &\triangleq p.x := e;\ \wr C' \wr\\
\wr\ \text{if } p.e \text{ then } C_1 \text{ else } C_2;\ C'\ \wr &\triangleq \text{if } p.e \text{ then } \wr\ C_1\ \wr \text{ else } \wr\ C_2 \wr;\ \wr C' \wr\\
\wr\ X(\vec{p}\,);\ C'\ \wr &\triangleq X(\vec{p}\,);\ \wr C' \wr\\
\wr\ q : X(\vec{p}\,).C;\ C'\ \wr &\triangleq \wr C' \wr
\end{aligned}$$

It is easy to verify how, for any $C$, $\lceil \wr C \wr \rceil$ is well-defined as by definition 4.2.

We extend the operator $\wr \cdot \wr$ on *procedure context*, as follows:

$$\wr \mathscr{C} \wr \triangleq \{X(\vec{p}) = \wr C \wr \mid X(\vec{p}) = C \in \mathscr{C}\}$$

We now prove that this operator is complete with respect to the semantics: for each $C$ such that

$$\langle C, \Sigma, \mathscr{C}\rangle \Downarrow^{M} \Sigma'$$

There exists $M^*$ such that:

$$\langle \wr C \wr, \Sigma, \wr \mathscr{C} \wr\rangle \Downarrow^{M^*} \Sigma' \tag{4.5}$$

**Proof:**
We proceed by induction on the size of $M$

- *size: 0* In this case the $C = \mathbf{0}$, by definition $\wr C \wr = \mathbf{0}$, thus by definition of $\cdot \Downarrow^{\cdot} \cdot$ the conclusion holds.

- *size: n+1* In this case we have two hypothesis:

$$\langle C, \Sigma, \mathscr{C} \rangle \xrightarrow{\mu} \langle C', \Sigma', \mathscr{C} \rangle \tag{H1}$$

$$\exists M',\ \langle \wr C' \wr, \Sigma', \wr \mathscr{C} \wr \rangle \Downarrow^{M'} \Sigma'' \tag{IH}$$

  By inversion of (H1) we find that $C = I; C^{\bigstar}$, we proceed by cases on the structure of $I$:

  - *assign, communication*: By further inversion of (H1) we find $C^{\bigstar} = C'$.
    We now find:
    $$\wr C \wr = \wr I; C^{\bigstar} \wr = \wr I; C' \wr = I;\ \wr C' \wr$$
    Using (IH) we construct $M^* = \mu :: M'$
  - *selection, runtime term*: By further inversion of (H1) we find $C^{\bigstar} = C'$ and $\Sigma = \Sigma'$.
    We now find:
    $$\wr C \wr = \wr I; C^{\bigstar} \wr = \wr I; C' \wr = \wr C' \wr$$
    Using (IH) we construct $M^* = M'$
  - *function call*: By further inversion of (H1) we find $X(\vec{p}) = C_X \in \mathscr{C}$, $C' = p_2 : X(\vec{p}\,).C^{\bigstar}; \ldots; p_n : X(\vec{p}\,).C^{\bigstar}; C_X[\vec{q}/\vec{p}\,] \fatsemi C^{\bigstar}$ and $\Sigma = \Sigma'$
    We now find:
    $$\wr C \wr = \wr X(\vec{p}); C^{\bigstar} \wr = X(\vec{p}); \wr C^{\bigstar} \wr$$
    Which can do a $\mu$ step of the operational semantics and reach:
    $$p_2 : X(\vec{p}\,).C^{\bigstar}; \ldots; p_n : X(\vec{p}\,).C^{\bigstar}; \wr C_X \wr\, [\vec{q}/\vec{p}\,] \fatsemi \wr C^{\bigstar} \wr$$
    Which can do $n-1$ steps $\tau @ p_2, \ldots, \tau @ p_n$ and reach
    $$\wr C_X \wr\, [\vec{q}/\vec{p}\,] \fatsemi \wr C^{\bigstar} \wr$$
    We now look at $C'$ and notice, by reasoning on the definition of *sequantial composition* and *process substitution* (proven in Appendix A):
    $$\wr C' \wr = \wr C_X[\vec{q}/\vec{p}\,] \fatsemi C^{\bigstar} \wr = \wr C_X[\vec{q}/\vec{p}\,] \wr \fatsemi \wr C^{\bigstar} \wr = \wr C_X \wr\, [\vec{q}/\vec{p}\,] \fatsemi \wr C^{\bigstar} \wr$$
    Using (IH) we construct
    $$M^* = \mu :: \tau @ p_2 :: \ldots :: \tau @ p_n :: M'$$

– *conditional* (for brevity we will only consider the *true* case, the other one follows by symmetry): By further inversion of (H1) we find $C' = C_1 \fatsemi C^\bigstar$ and $\Sigma = \Sigma'$.
We now find:

$$\wr C \wr = \wr \text{ if } p.e \text{ then } C_1 \text{ else } C_2;\ C^\bigstar \wr = \text{if } p.e \text{ then } \wr\, C_1 \wr \text{ else } \wr\, C_2 \wr;\ \wr C^\bigstar \wr$$

Which can do a $\mu$ step and reach:

$$\wr C_1 \wr \fatsemi \wr\, C^\bigstar \wr = \wr C_1 \fatsemi C^\bigstar \wr = \wr C' \wr$$

Using (IH) we construct $M^* = \mu :: M'$

**Ramifications on the semantic for procedure calls**

Since, as we saw in the previous section, *runtime terms* are irrelevant for the computation, we will now change the semantics for CALL-FIRST to the following rule:

$$\frac{X(\vec{q}\,) = C \in \mathscr{C}}{\langle X(\vec{p}\,); C', \Sigma, \mathscr{C}\rangle \dot{\to} \langle C[\vec{q}/\vec{p}\,] \fatsemi C', \Sigma, \mathscr{C}\rangle}\ \textsc{call-first}$$

The correctness of this change follows from the previous proof, since both rules have as result $\wr$-equivalent configurations (we will omit a formal discussion of this equivalence) the state reached by the computation is unchanged.
The justification for not specifying the transition label associated with the inference rule is that we are going to mostly ignore the transition labels from now on.

**Completeness of the instrumented semantics**

We take a $C, \mathscr{C}$ for which $\lceil\cdot\rceil$ is well defined. We have to prove that, given:

$$\langle C, \lfloor\Sigma\rfloor, \mathscr{C}\rangle \Downarrow^M \Sigma'$$

then there exists $\Sigma''$ such that:

$$\langle \lceil C\rceil, \Sigma, \lceil\mathscr{C}\rceil\rangle \Downarrow \Sigma'' \wedge \Sigma' = \lfloor\Sigma''\rfloor$$

**Proof:**
TODO: fai dimostrazione completa

**Putting all the steps together**

TODO: intuizione di perchè funziona

### 4.3.2 Completeness of type extension

This lemma creates tells us *lifting* a choreography maintains typing.
Formally:

$$\Delta; \Gamma; \bot \vdash C \Rightarrow \Delta; \Gamma; \bot \vdash \lceil \wr C \wr \rceil$$

### 4.3.3 Preservation lemma

The preservation lemma (i.e. subject reduction) states that the operational semantics preserves typing, that is *starting from a well typed configuration and executing a step of the semantics gives you a well typed configuration.* We are now working only in the instrumented semantics, thus we will omit the redundant brackets when sufficently clear.
Given $C$ in [**Chor**], $\Sigma$ in [**CStore**], a configuration $\langle C, \Sigma, \mathscr{C} \rangle$ is well-typed for a $pc \in \mathscr{L}$ when $\Delta, \Gamma, pc \vdash C$ and $\Gamma \vdash \Sigma$

Thus the preservation lemma is stated as follows:

$$\Delta, \Gamma, pc \vdash C \wedge \Gamma \vdash \Sigma \wedge \langle C, \Sigma, \mathscr{C} \rangle \rightarrow \langle C', \Sigma', \mathscr{C} \rangle \Rightarrow \Delta, \Gamma, pc \vdash C' \wedge \Gamma \vdash \Sigma' \tag{4.6}$$

It is easy to see how the preservation lemma can be easily carried over to $\cdot \twoheadrightarrow \cdot$.

### 4.3.4 Unwinding lemma

The unwinding lemma states that the operational semantics preserves low-equal configurations. We formalize this as follows:
Given well-typed $C_1$ $C_2$ in [**Chor**], well-formed $\Sigma_1, \Sigma_2$ is [**CStore**] then:

$$C_1 \approx_{low} C_2 \wedge \Sigma_1 \approx_{low} \Sigma_2 \wedge \langle C_1, \Sigma_1, \mathscr{C} \rangle \rightarrow \langle C_1', \Sigma_1', \mathscr{C} \rangle$$

implies that there exists $C_2'$ $\Sigma_2'$ s.t.

$$\langle C_2, \Sigma_2, \mathscr{C} \rangle \twoheadrightarrow \langle C_2', \Sigma_2', \mathscr{C} \rangle \wedge C_1' \approx_{low} C_2' \wedge \Sigma_1' \approx_{low} \Sigma_2' \tag{4.7}$$

## 4.4 Main proof

We remind ourself the main theorem as defined in (4.1):

$$\begin{aligned} & \Delta; \Gamma; \bot \vdash C \Rightarrow \Sigma_1 \equiv_{low}^{\Gamma} \Sigma_2 \\ \Rightarrow & \langle C, \Sigma_1, \mathscr{C} \rangle \Downarrow^{M_1} \Sigma_1' \Rightarrow \ \langle C, \Sigma_2, \mathscr{C} \rangle \Downarrow^{M_2} \Sigma_2' \\ \Rightarrow & \Sigma_1' \equiv_{low}^{\Gamma} \Sigma_2' \end{aligned}$$

**Proof:**
By completeness (4.4) we find:

$$\langle \lceil \wr C \wr \rceil, \lceil \Sigma_1 \rceil^\Gamma, \mathscr{C} \rangle \Downarrow [\Sigma]'_1 \quad \text{and} \quad \langle \lceil \wr C \wr \rceil, \lceil \Sigma_2 \rceil^\Gamma, \mathscr{C} \rangle \Downarrow [\Sigma]'_2$$

such that

$$\lfloor [\Sigma]'_1 \rfloor = \Sigma'_1 \quad \text{and} \quad \lfloor [\Sigma]'_2 \rfloor = \Sigma'_2 \tag{4.8}$$

By extension of typing 4.3.2 we have:

$$\Delta; \Gamma; \bot \vdash \lceil \wr C \wr \rceil$$

By definition of $\lceil \cdot \rceil^\Gamma$ we have

$$i \in 1, 2 \quad \lceil \Sigma_i \rceil^\Gamma \Rightarrow \Gamma \vdash \lceil \Sigma_i \rceil^\Gamma$$

We can thus use preservation (4.6) to find

$$\Gamma \vdash [\Sigma]'_1 \quad \text{and} \quad \Gamma \vdash [\Sigma]'_2$$

That, with 4.8 let's us reduce the proof of $\Sigma'_1 \equiv^\Gamma_{low} \Sigma'_2$ to the proof of $[\Sigma]'_1 \approx_{low} [\Sigma]'_2$
We are now reduced the proof to the following:

$$\begin{aligned} & \Delta; \Gamma; \bot \vdash \lceil \wr C \wr \rceil \\ \Rightarrow & \langle \lceil \wr C \wr \rceil, \lceil \Sigma_1 \rceil^\Gamma, \mathscr{C} \rangle \Downarrow [\Sigma]'_1 \\ \Rightarrow & \langle \lceil \wr C \wr \rceil, \lceil \Sigma_2 \rceil^\Gamma, \mathscr{C} \rangle \Downarrow [\Sigma]'_2 \\ \Rightarrow & [\Sigma]'_1 \approx_{low} [\Sigma]'_2 \end{aligned}$$

By $\Sigma_1 \equiv^\Gamma_{low} \Sigma_2$ and well-formedness of the contexts we have

$$\lceil \Sigma_1 \rceil^\Gamma \approx_{low} \lceil \Sigma_2 \rceil^\Gamma$$

By reflexivity of $\approx_{low}$ we have

$$\lceil \wr C \wr \rceil \approx_{low} \lceil \wr C \wr \rceil$$

We generalize over the specific construction of $\lceil \Sigma_1 \rceil^\Gamma$, $\lceil \Sigma_2 \rceil^\Gamma$, $\lceil \wr C \wr \rceil$ and just keep the previously stated low-equivalences between them. At this point the proof follows by induction on the size of the first transitions:

- *size 0*: Transition of size zero means that the first choreography is $\mathbf{0}$. By definition of $\cdot \approx_{low} \cdot$ the second is $\mathbf{0}$ as-well. By hypothesis the two $\Sigma$ are equivalent, thus the conclusion follows
- *size n+1*: This means that the first transition is composed by at least one step of the operational semantics.

  $$\langle [C]_1, [\Sigma]_1, \mathscr{C} \rangle \to \langle [C]^*_1, [\Sigma]^*_1, \mathscr{C} \rangle$$

  By unwinding (4.7) we find $[C_2]^*$, $[\Sigma]^*_2$ s.t. the low-equivalence is respected. By preservation (4.6) we find the hypothesis of well-typedness and well-formedness needed to use the induction hypothesis, with which we prove the conclusion.

# Appendices

# Appendix A

# $\wr \cdot \wr$ preserved by syntactic transformations

**TODO: finire, ho dimostrazione sketch su carta**

## A.1 Sequential composition

We need to prove the following:

$$\wr C_1 \fatsemi C_2 \wr = \wr C_1 \wr \fatsemi \wr C_2 \wr$$

## A.2 Process substitution

We need to prove the following:

$$\wr C[\vec{q}/\vec{p}] \wr = \wr C \wr [\vec{q}/\vec{p}]$$

# Bibliography


[1] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying Constant-Time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, Austin, TX, August 2016. USENIX Association.

[2] Hendrik P Barendregt et al. *The lambda calculus*, volume 3. North-Holland Amsterdam, 1984.

[3] Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. *Theoretical Computer Science*, 802:38–66, 2020.

[4] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[5] Joseph A Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11. IEEE, 1982.

[6] Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In *Software safety and security*, pages 319–347. IOS Press, 2012.

[7] International Organization for Standardization. ISO/IEC 27001:2022. `https://www.iso.org/standard/82875.html`, 2022. Accessed: 2025-07-11.

[8] Gilles Kahn. Natural semantics. In *Annual symposium on theoretical aspects of computer science*, pages 22–39. Springer, 1987.

[9] Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, 2023.

[10] Fabrizio Montesi. Introduction to choreographies, 2023. Available at: `https://www.fabriziomontesi.com/introduction-to-choreographies/`. Accessed: 2025-07-08.

[11] Andrew Myers. Proving noninterference for a while-language using small-step operational semantics. 2011.

[12] Minh Ngo, Frank Piessens, and Tamara Rezk. Impossibility of precise and sound termination-sensitive security enforcements. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 496–513. IEEE, 2018.

[13] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications*, volume 104. Springer, 1992.

[14] Gordon D Plotkin. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60:3–15, 2004.

[15] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, 1953.

[16] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[17] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of computer security*, 4(2-3):167–187, 1996.

[18] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proceedings 10th Computer Security Foundations Workshop*, pages 156–168. IEEE, 1997.

[19] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[20] Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

[21] Xichen Zhang, Mohammad Mehdi Yadollahi, Sajjad Dadkhah, Haruna Isah, Duc-Phong Le, and Ali A Ghorbani. Data breach: analysis, countermeasures and challenges. *International Journal of Information and Computer Security*, 19(3-4):402–442, 2022.

# Ringraziamenti

Grazie a tutti