ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Scuola di Scienze

Corso di Laurea Magistrale in Informatica

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

Relatore:
Saverio Giallorenzo

Presentata da:
Marco Bertoni

Correlatore:
Marco Peressotti

Sessione Ottobre 2025
Anno Accademico 2024/2025

The method of 'postulating' what we want has many advantages; they are the same as the advantages of theft over honest toil.

Russel - 1919

Abstract

TODO: fare alla fine

Contents

1 Introduction **1**

2 Background **3**
2.1 Choreographies . . . 3
2.1.1 Processes . . . 3
2.1.2 Syntax . . . 4
2.1.3 Semantics . . . 5
2.1.4 Notes on expressivity . . . 10
2.1.5 Notes on small-step semantics . . . 10
2.2 Information flow analysis . . . 10
2.3 Non-Interference . . . 12
2.3.1 Definition . . . 13
2.4 Lean Proof Assistant . . . 15
2.4.1 Underlying theory . . . 15
2.4.2 Lean in a nutshell . . . 15

3 Enforcement of the Non-Interference Property in Choreographies **17**
3.1 Definition of the flow-policy . . . 18
3.2 Intuitive presentation of the type-system . . . 18
3.2.1 A motivating example for Program Counter labeling . . . 18
3.2.2 Typing procedure calls . . . 19
3.2.3 Putting it all together . . . 19
3.3 Formal definition of the Type System . . . 20
3.3.1 Judgment relation . . . 20
3.3.2 Typing Rules for Expressions . . . 20
3.3.3 Typing Rules for Instructions . . . 21
3.3.4 Typing Rules for Choreographies . . . 22

4 Soundness of the Type System **23**
4.1 Overall Statement and Proof Obligations . . . 23
4.2 Instrumented Choreographies . . . 26

4.2.1 Syntax . . . 26
4.2.2 Lowering and Lifting of [Chor] . . . 27
4.2.3 Low Equivalence of [Chor] . . . 28
4.2.4 Well-Formedness of [**CStore**] . . . 29
4.2.5 Correctness of [**CStore**] . . . 29
4.2.6 Low Equivalence on [**CStore**] . . . 30
4.2.7 Semantics . . . 30
4.2.8 Extension of the Type System . . . 32
4.3 Auxiliary Lemmas . . . 33
4.3.1 Completeness Lemma . . . 33
4.3.2 Completeness of Type Extension . . . 40
4.3.3 Preservation Lemma . . . 41
4.3.4 Unwinding Lemma . . . 42
4.4 Main Proof . . . 42

5 Construction of Δ 45
5.1 Context Reconstruction Algorithm . . . 45
5.1.1 Local Expression Reconstruction . . . 46
5.1.2 Choreography Reconstruction . . . 47
5.2 Proof of Well-Typedness . . . 48
5.2.1 Creating δ . . . 48
5.2.2 One step soundness . . . 49
5.2.3 Well-Typed δ . . . 51
5.3 Proof of pc Subsumption . . . 51
5.4 Termination . . . 52
5.5 Extension to Full Type Inference . . . 52

6 Lean mechanization 53
6.1 Naming Conventions . . . 53
6.2 Imported Definitions and Results . . . 53
6.2.1 Encoding of $\mathscr{L}$ . . . 53
6.2.2 Reference choreographies . . . 53
6.3 Project Structure . . . 54
6.4 Coverage of the Mechanization . . . 56
6.5 An Illustrative Example . . . 56

Appendices 63

Appendix A Proofs on Syntactic Transformations 65
A.1 Sequential Composition and $\wr\cdot\wr$ . . . 65
A.2 Process Substitution and $\wr\cdot\wr$ . . . 65

A.3 Store Update and $\lfloor \cdot \rfloor$ 65
A.4 Process Substitution and $\lceil \cdot \rceil$ 66

Appendix B Decomposition of Sequential Composition Execution 67
B.1 **[Chor]** Sequential Composition 67
B.2 **Chor** Sequential Composition 68

Appendix C Helper Lemmas for Constraint Reconstruction 69
C.1 Monotonicity of $\phi_{\mathscr{C}}$ 69
C.2 Rewriting η in cansolve 69
C.3 Substitution in cansolve 70

Chapter 1

Introduction

TODO: fare alla fine

Chapter 2

Background

2.1 Choreographies

Unless otherwise indicated, all content in this chapter is derived from: *Introduction to Choreographies [21]* by *Fabrizio Montesi*, updated with the errata corrige present in Montesi's website [22].

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 $X(\vec{p}\,) = C$, 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[1].

- 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* $\boldsymbol{p \to q[L]}$. Label selections are required for the correct coordination of distributed branching, and solve a problem known as *knowledge of choice* [6]. Since the discussion of this problem 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 e, 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

[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

conditional **if $p.e$ then C_1 else 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*.

- 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 run time term $\boldsymbol{X(\vec{p}).C}$, The key motivation behind the introduction of run time terms is to correctly represent distributed recursion. They syntactically denote the intermediate states originating from the independent procedure calls executed by the processes participating in the choreography. The discussion of this topic goes beyond the scope of this work, we will just treat run time terms as arbitrary instruction of the language.

- e denotes a *local 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 [27], forming a Labeled 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. Σ represents the *choreographic store*. Let us now define it.

A *process store* σ 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 σ 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**) Σ 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 Σ 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** σ, 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 σ*. 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:

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

Process Names of a Choreography

Before defining the *transition relation*, we need to 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 argument.

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

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

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* $\dot{\to}$ as the smallest relation derived by the following inference schemata:

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

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

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

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

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

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

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

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

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

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 and can be found in Montesi's book [21]. The main intuition is the following: processes are independent and can *enter* the procedure at any interleaved order, but we need some syntactical marker in the choreography to correctly coordinate distributed recursion. 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 into 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.

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

2.1.4 Notes on expressivity

The presented language can be shown to be Turing Complete [8]. For Turing-complete languages, any *nontrivial extensional* (i.e., semantic) property of programs is undecidable [28]. 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 [16]. With the proliferation of interconnected systems, sensitive data such as personal information, financial records, and classified communications is constantly processed, transmitted, and stored [35]. 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 [35]. Information Flow Analysis [11] 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 [11], secure information flow means that all data transfers 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 run time 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 [29]. *Side channels* (e.g., timing, termination, resource usage, cache effects, message sizes, or scheduler-dependent behavior) [18] 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 [2].

2.3 Non-Interference

Denning's work [11] 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 [12] 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 [31]. 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 [31].

In contrast, purely dynamic enforcement mechanisms such as run time monitors are unable to detect certain classes of implicit information leaks [29]. 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 `public`, 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

[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* [29] is accepted as result of the analysis.

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 branches not taken. Since information flow security is a property of all possible executions [29] [11], 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 [31] and thus offer a more precise and rigorous framework for enforcing confidentiality.

2.3.1 Definition

Let us consider a simple imperative programming language [33] with commands such as assignments, sequencing, conditionals, and loops. A program state s is typically modeled as a mapping from variables to values [33], 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.

The semantics of a program C is given [17] [25] 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 [31] [12], for all $s_1, s_2 \in S$:

$$s_1 \equiv_L s_2 \Rightarrow [\![C]\!](s_1) \approx_L [\![C]\!](s_2)$$

Here, $\approx_L$ denotes *observational equivalence*[4] from the perspective of a low-security observer. In a termination sensitive setting [14], 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.

[4] This notion of observational equivalence can be naturally extended to account for additional observables beyond final low-security state and termination behavior [29]. 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).

Termination Sensitivity

When formalizing non-interference, a key consideration is whether termination behavior should be treated as an observable effect [14]. 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 [32]. 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
```

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 [12]. Under this weaker definition, the observational equivalence relation $\approx_L$ only requires that whenever two executions terminate, they yield indistinguishable low-observable states [31]:

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

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 [24].

2.4 Lean Proof Assistant

A *proof assistant* is an interactive system for developing machine-checked mathematics. Mechanized proofs[5] provide stronger assurance by forcing full formalization and having a small trusted checker verify every inference, catching subtle gaps and mistakes that pen-and-paper often miss. They also yield reproducible, maintainable, and scalable artifacts (proof scripts) that anyone can re-run, audit, and extend with automation, making it feasible to verify large, evolving systems. [13] These guarantees are particularly useful for meta-theoretic results about programming languages, where small omissions (e.g. missing cases in an induction) are common and subtle. [4]

2.4.1 Underlying theory

Lean is based on a version of dependent type theory known as the Calculus of Constructions [7], with a countable hierarchy of non-cumulative universes and inductive types [3]. Dependent type theory is a powerful and expressive language allowing you to express complex mathematical assertions.
Lean employs a cumulative hierarchy $Type_0 \subset Type_1 \subset \cdots$ (universe polymorphism) to avoid paradoxes while retaining expressivity [19].

Under the Curry-Howard correspondence, propositions are types and proofs are programs inhabiting those types [15].
Using the Lean programming language we are able to write proof-terms that inhabit arbitrary propositions by type-checking. An inhabited proposition is considered as true. This view of the Curry-Howard correspondence gives us *proof irrelevance*, that is two proof-terms are considered equal if they construct the same type.

Proofs are usually written by the user in an higher-level, *tactics* language and then compiled to proof-terms. As we said, 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. Soundness of the proof ultimately rests on a small *trusted kernel*: a compact type-checker that implements Lean's dependent type theory and accepts only well-typed declarations up to definitional equality. [10]
The compilation step from tactics to

2.4.2 Lean in a nutshell

Types, terms, and equality. Lean 4 is a dependent type theory with a distinguished universe `Prop` for logical propositions. We distinguish *definitional equality* (judgmental, i.e. equal by computation) from *propositional equality* (an inhabitant

[5] In this thesis, "mechanized" means that all the stated theorems have corresponding Lean artifacts that type-check.

of `a = b`). Most rewriting in this work uses propositional equality via standard congruence and rewriting principles.

Inductive types and inductive relations. Abstract syntax trees (ASTs), typing rules, and small-step semantics are represented as *inductive* declarations. Crucially, judgments like typing and transition relations are encoded as *inductive predicates* (relations) whose constructors correspond one-to-one with the inference rules in this manuscript. Induction over derivations ("rule induction") is the default proof method for meta-properties.

Pattern matching, recursion, and mutuality. Lean supports structural recursion and induction; when definitions depend on each other (e.g. instructions and choreographies), we use *mutual* recursion/induction. This mirrors the mutual inductive structure of the language and will reappear whenever we prove properties by simultaneous induction on related syntactic categories.

Type classes and algebraic structure. Algebraic structure (orders, joins, bottom) is provided via *type classes*. For the security lattice $\mathscr{L}$ we rely on instances such as partial orders and (join-)semilattices with a least element ($\bot$). Type-class resolution lets us write generic lemmas that work for any instance of these interfaces (e.g. monotonicity of joins, distributivity properties actually used in typing).

Finite collections and decidability. We use `Finset` to represent finite sets (e.g. sets of process names). Membership and set operations are computable; many lemmas require decidable equality on elements. This setup simplifies counting arguments, case splits over finite supports, and proofs that manipulate environments whose keys form finite domains.

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 [31] 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 Π, every program that is well-typed under Π is *semantically compliant* with Π; 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 [23] [34], 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 3.3 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 $\bot$ such that every $l \in \mathscr{L}$ respects $\bot \sqsubseteq l$. These labels capture Denning's notion of security classes where every object manipulated by the program has an associated security class. We implement this by assigning a security label to every variable of the program.

A *process security labeling* γ models the security class *i.e., security label* associated with the variables accessed by a process. Formally, a process labeling is a function from variables to security labels:

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

We write **SecPLab** for the set of all process security labelings. Similarly to what we defined for *choreographic stores*, we define a *choreographic security labeling* Γ as a map from process names to their respective process labeling. Formally,

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

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

In the non-interference framework, the flow-policy is fully specified by the security labeling: 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 labeling

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 this program would follow the flow-policy Γ 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 and 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 $p.a$. We can, thus, update our 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 [29] builds type-systems for a *while language*, but we find ourselves 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, pc such that $\Gamma\ \in\ \Delta\ X\ pc$, then the body of X is *well typed* under Γ and pc.

Further discussion on how to compute the context which carries this property will follow in Chapter 5, 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 Δ, 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 $\ell \in \mathscr{L}$

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

$$\gamma \vdash v : \bot$$

Variable

$$\gamma \vdash x : \gamma\ x$$

N-ary function

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

We assume primitive functions to be label-preserving and to 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 arguments.

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.

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 Δ, Γ' 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 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 Γ by updating the context pointed by each p_i to mirror the one pointed by q_i. Formally (in the scalar case):

$$\Gamma[q \mapsto p]\ r \triangleq \begin{cases} \Gamma\ p & \text{if } q = r \\ \Gamma\ r & \text{otherwise} \end{cases} \tag{3.1}$$

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

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

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 its 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 labeled $\sqsubseteq low$ in Γ.

The soundness theorem states that *well-typed programs satisfy termination insensitive non-interference*:

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

To fully specify this theorem, we need to define an equivalence relation that encodes the concept of *low-equivalence under termination insensitive non-interference*. To join this need with the semantics presented in 2.1, we introduce a *natural semantics* [17] for choreographies. We define a relation:

$$\langle C, \Sigma, \mathscr{C} \rangle \Downarrow^M \Sigma' \tag{4.1}$$

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:

$$\begin{aligned} & \Delta; \Gamma; \bot \vdash C \Rightarrow \Sigma_1 \equiv^{\Gamma}_{low} \Sigma_2 \\ \Rightarrow & \langle C, \Sigma_1, \mathscr{C} \rangle \Downarrow^{M_1} \Sigma_1' \Rightarrow \ \langle C, \Sigma_2, \mathscr{C} \rangle \Downarrow^{M_2} \Sigma_2' \\ \Rightarrow & \Sigma_1' \equiv^{\Gamma}_{low} \Sigma_2' \end{aligned} \tag{4.2}$$

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 Δ for typing procedure calls was explained previously.

Context subsumption We say that Δ satisfies *context subsumption* if lowering the control level always maintains typing:

$$\Gamma \in \Delta\ X\ pc\ \wedge\ pc' \sqsubseteq pc \implies \Gamma \in \Delta\ X\ pc' \tag{4.3}$$

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 $X(\vec{p}) = C \in \mathscr{C}$,

$$\mathrm{pn}(C) \subseteq \{\vec{p}\}. \tag{4.4}$$

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 [21]. 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 Δ to be consistent with the declaration context $\mathscr{C}$: every declared procedure must typecheck under every security environment that Δ admits for it (at any program counter). Formally, for every clause $X(\vec{p}) = C \in \mathscr{C}$ and every $pc \in \mathscr{L}$,

$$\Delta\ X\ pc = \mathcal{G} \implies \forall \Gamma \in \mathcal{G}.\ \Delta; \Gamma; pc \vdash C \tag{4.5}$$

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 Γ 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 Δ-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 [5]: 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 we avoid spurious name capture when instantiating X with actual participants (e.g., at call sites or during inlining). We will use α-renaming implicitly to maintain this invariant.

Properties of expressions

The semantics referenced [21] 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 σ, 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 non-determinism that a low observer could notice.

4.2 Instrumented Choreographies

A natural strategy to prove the main theorem stated in (4.2) is to use induction on the length of the computation. Unfortunately, this approach is not sufficient: 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 Σ_1, Σ_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 choreographies 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 brackets as being *not observable* by a participant of level $l \sqsubseteq$ *low*. We call this participant 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.

$$
\begin{array}{rl}
p, q \in \mathsf{Pid} & \text{(process names)} \\
x \in \mathsf{Var} & \text{(local variables)} \\
X \in \mathsf{ProcName} & \text{(procedure names)} \\
L \in \mathsf{Label} & \text{(selection labels)} \\
e \in \mathsf{Expr} & \text{(expressions)}
\end{array}
$$

Choreographies:

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

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 length (possibly zero), thus we want to be able to denote as different terms, for example, $[\mathbf{0}]$ and $\mathbf{0}$.
- *Removal of the runtime term and label selection instructions:* This is not necessary but it simplifies the soundness proof. Its justification is that both instructions have no impact on the final computed stores and are needed by choreographies to deal with projection [21], 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 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 its shape.

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

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

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*, defined by the following equation:

$$C = \lfloor \lceil C \rceil \rfloor \tag{4.6}$$

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:

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

$$X(\vec{p}) \;\approx_{\mathsf{low}}\; X(\vec{p})$$

By construction, $\approx_{\mathsf{low}}$ is an equivalence relation 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 Γ (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 Γ. That is, it will choose bracketing of values depending on the security label in Γ 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 its reference counterpart. Formally, given a **[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 occurrence of v has to be considered as either a **Val** object or a **[Val]** object which is *not bracketed*. Every occurrence of $[v]$ is a *bracketed* **[Val]**. We will use $\mathbf{v}$ when it is not specified whether 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 Σ when

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

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

4.2.6 Low Equivalence on [CStore]

We define a notion of low-equivalence on [**Val**]: Given ν_1, ν_2 then

$$\nu_1 \approx_{low} \nu_2 \quad \text{iff} \quad \begin{cases} \nu_1 = v_1 \text{ and } \nu_2 = v_2 \text{ and } v_1 = v_2 \\ \text{or} \\ \nu_1 = [v_1] \text{ and } \nu_2 = [v_2] \end{cases} \tag{4.7}$$

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

We fix an observation level $low \in \mathscr{L}$ and a choreographic security labeling Γ. 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 invariant 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 low.
- If the expression already evaluates to a bracketed value $[v]$, we propagate the bracket on write. This never *un-brackets* 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 un-bracketed 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 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 low and, via the assignment and communication rules, forces all subsequent effects to remain bracketed and thus invisible at low. Dually, when the guard evaluates to an *un-bracketed* boolean, the choice is already determined at low, so the corresponding un-bracketed 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 $\vdash f(\vec{v}) \downarrow \boldsymbol{\nu}$. The extension *carries* brackets from the arguments of the function to its returned values, that is: given a function

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

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

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:

$$\lfloor [\sigma] \rfloor \vdash e \downarrow v \Leftrightarrow ([\sigma] \vdash e \downarrow \boldsymbol{\nu} \wedge \lfloor \boldsymbol{\nu} \rfloor = v) \tag{4.8}$$

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 aswell.

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'.

4.3 Auxiliary Lemmas

To be able to prove the main non-interference theorem as defined in 4.2, we will firstly assume a few auxiliary lemmas. Since the majority of the proof approach is taken directly from [23], we will abstain from reproducing them in this work when sufficiently similar.

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:

$$\langle \lceil \wr C \wr \rceil, \lceil \Sigma \rceil^{\Gamma}, \lceil \mathscr{C} \rceil \rangle \Downarrow [\Sigma]' \wedge \lfloor [\Sigma]' \rfloor = \Sigma' \tag{4.9}$$

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 (*i.e.*, out-of-order): given the same configuration it can admit 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:

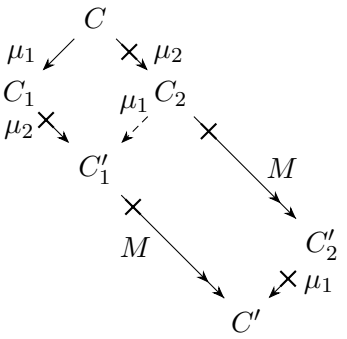
$$\frac{X(\vec{q}\,) = C \in \mathscr{C} \qquad \vec{p} = p_1, \ldots, p_n}{\langle X(\vec{p}\,); C', \Sigma, \mathscr{C} \rangle \xrightarrow{\tau @ p_1} \langle p_2 : X(\vec{p}\,).C'; \ldots; p_n : X(\vec{p}\,).C'; C[\vec{q}/\vec{p}\,] \fatsemi C', \Sigma, \mathscr{C} \rangle} \text{ CALL-FIRST}$$

It is to be noted how the new CALL-FIRST is a special case of the previous one. This leads us to see how deterministic transitions are a subset of the transitions admitted in the reference semantics.

The proof of completeness of the deterministic semantics follows from the work from Cruz-Filipe, Montesi, and Peressotti [9][1] which proves some important properties of the semantics, those being:

- *deadlock freedom:* Any choreography that is not **0** can execute.
- *diamond property:* Any two distinct one-step reductions from the same state are reconcilable: there exist further reductions from each successor that lead to an identical state.
- *convergence:* Any two executions of a choreography that end in a terminated choreography must finish in the same state.

Using the previously stated properties, we proceed by a graphical proof: 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. By the diamond property, there is a μ_1 from C_2 to C_1' (depicted as a dashed arrow). Since we can *compose diamonds*, we can repeat this step the zero or more times necessary to *wait* for μ_1 to be deterministic. Moreover no maximal computation can end with a non-deterministic transition (which is a trivial corollary of deadlock freedom) which gives us the existence of this transition. Thus, 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.



It is easy to see, by cases on the reference semantics, how, if from C there are no two different one-step reachable C_1, C_2, then the transition is in the deterministic subset.

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 on, 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, making us able to use freely the $\lceil\cdot\rceil$ operator. We introduce the

[1]The language and semantics defined in the paper have some trivial differences, yet it is easy to see how the proofs presented there can be carried over to the language and semantics used in this thesis.

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 4.6.

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:

$$\langle \wr C \wr, \Sigma, \wr \mathscr{C} \wr \rangle \Downarrow^{M^*} \Sigma' \tag{4.10}$$

Proof:
We proceed by induction on the size of M

- *size: 0* In this case the $C = \mathbf{0}$ and by definition $\wr C \wr = \mathbf{0}$, thus 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 $C = I; C^\star$, we proceed by cases on the structure of I:

  - *assign, communication*: By further inversion of (H1) we find $C^\star = C'$. We now find:

$$\wr C \wr = \wr I; C^\star \wr = \wr I; C' \wr = I;\ \wr C' \wr$$

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

– *selection, runtime term*: By further inversion of (H1) we find $C^\star = C'$ and $\Sigma = \Sigma'$.
We now find:

$$\wr C \wr = \wr I; C^\star \wr = \wr I; C' \wr = \wr C' \wr$$

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

– *function call*: By further inversion of (H1) we find $X(\vec{p}) = C_X \in \mathscr{C}$, $C' = p_2 : X(\vec{p}\,).C^\star; \ldots; p_n : X(\vec{p}\,).C^\star; C_X[\vec{q}/\vec{p}\,] \fatsemi C^\star$ and $\Sigma = \Sigma'$
We now find:

$$\wr C \wr = \wr X(\vec{p}); C^\star \wr = X(\vec{p}); \wr C^\star \wr$$

Which can do a μ step of the operational semantics and reach:

$$p_2 : X(\vec{p}\,).C^\star; \ldots; p_n : X(\vec{p}\,).C^\star; \wr C_X \wr [\vec{q}/\vec{p}\,] \fatsemi \wr C^\star \wr$$

Which can do $n-1$ steps $\tau @ p_2, \ldots, \tau @ p_n$ and reach

$$\wr C_X \wr [\vec{q}/\vec{p}\,] \fatsemi \wr C^\star \wr$$

We now look at C' and notice, by reasoning on the definition of *sequential composition* and *process substitution* (proven in Appendix A):

$$\begin{aligned} \wr C' \wr &= \wr p_2 : X(\vec{p}\,).C^\star; \ldots; p_n : X(\vec{p}\,).C^\star; C_X[\vec{q}/\vec{p}\,] \fatsemi C^\star \wr \\ &= \wr C_X[\vec{q}/\vec{p}\,] \fatsemi C^\star \wr = \wr C_X[\vec{q}/\vec{p}\,] \wr \fatsemi \wr C^\star \wr = \wr C_X \wr [\vec{q}/\vec{p}\,] \fatsemi \wr C^\star \wr \end{aligned}$$

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 $C' = C_1 \fatsemi C^\star$ and $\Sigma = \Sigma'$.
We now find:

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

Which can do a μ step and reach:

$$\wr C_1 \wr \fatsemi \wr C^\star \wr = \wr C_1 \fatsemi C^\star \wr = \wr C' \wr$$

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, which also makes CALL-ENTER unreachable:

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

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 Σ'' such that:

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

Proof:
We proceed by induction of the size of $\cdot \Downarrow^M \cdot$:

- *size: 0*: We have
$$C = \mathbf{0} = \lceil C\rceil \quad \text{and} \quad \lfloor\Sigma\rfloor = \Sigma'$$
so the conclusion follows with $\Sigma'' = \Sigma$

- *size: n + 1*: We have the following hypothesis:
$$\langle C, \lfloor\Sigma\rfloor, \mathscr{C}\rangle \dot{\rightarrow} \langle C^\star, \Sigma^\star, \mathscr{C}\rangle \tag{H1}$$
$$\langle C^\star, \Sigma^\star, \mathscr{C}\rangle \Downarrow^{\cdot} \Sigma' \tag{H2}$$
$$\forall\ \Sigma^\heartsuit, \lfloor\Sigma^\heartsuit\rfloor = \Sigma^\star \Rightarrow \langle\lceil C^\star\rceil, \Sigma^\heartsuit, \lceil\mathscr{C}\rceil\rangle \Downarrow \Sigma'' \wedge \lfloor\Sigma''\rfloor = \Sigma' \tag{IH}$$
We proceed by inversion on (H1)
  - LOCAL: Inversion gives us:
$$C = p.x := e;\ C^\star \tag{I1}$$
$$\lfloor\Sigma\rfloor\ p \vdash e \downarrow v \tag{I2}$$
$$\Sigma^\star = \lfloor\Sigma\rfloor[p.x \mapsto v] \tag{I3}$$

By *correctness* of the local evaluation (4.8) and (I2) we find:

$$\Sigma\ p \vdash e \downarrow [v] \quad \vee \quad \Sigma\ p \vdash e \downarrow v \tag{4.11}$$

We proceed by cases on the hypothesis:

* *case left:* By definition of $\lceil\cdot\rceil$ we have:

  $$\lceil C \rceil = p.x := e;\ \lceil C^\star \rceil$$

  We do the following execution steps:

  $$\begin{aligned}&\langle p.x := e;\ \lceil C^\star \rceil, \Sigma, \lceil \mathscr{C} \rceil\rangle\\ &\to \langle \mathbf{0};\ \lceil C^\star \rceil, \Sigma[p.x \mapsto [v]], \lceil \mathscr{C} \rceil\rangle\\ &\to \langle \lceil C^\star \rceil, \Sigma[p.x \mapsto [v]], \lceil \mathscr{C} \rceil\rangle\end{aligned}$$

  From which, using the fact that (equivalences on syntactic transformations are proved in Appendix A)

  $$\lfloor \Sigma^\heartsuit \rfloor = \lfloor \Sigma[p.x \mapsto [v]] \rfloor = \lfloor \Sigma \rfloor [p.x \mapsto v] = \Sigma^\star$$

  We can use the induction hypothesis (IH)

* *case right:* We have two possible rules regarding assignment with (4.11) as antecedent, depending on the value of $\Gamma\ p\ x$. We proceed by *law of excluded middle* and find:

  $$\Gamma\ p\ x \not\sqsubseteq low \quad \vee \quad \Gamma\ p\ x \sqsubseteq low \tag{4.12}$$

  In the *left case*, we proceed exactly as the previously specified proof case, adjusting the inference rule used for the first step. In the *right case*, we follow similarly, with the only difference being:

  $$\Sigma^\heartsuit = \Sigma[p.x \mapsto v]$$

– COM: The inductive case follows similarly to the LOCAL one, substituting $p.x$ with $q.x$ when needed.

– COND-THEN: Inversion gives us:

$$\begin{aligned}&C = \text{if } p.e \text{ then } C_1 \text{ else } C_2;\ C' && \text{(I1)}\\ &C^\star = C_1 \fatsemi C' && \text{(I2)}\\ &\lfloor \Sigma \rfloor\ p \vdash e \downarrow \mathit{true} && \text{(I3)}\\ &\Sigma^\star = \lfloor \Sigma \rfloor && \text{(I4)}\end{aligned}$$

By definition of $\lceil\cdot\rceil$ we have:

$$\lceil C\rceil = \text{if } p.e \text{ then } \lceil C_1\rceil \text{ else } \lceil C_2\rceil;\ \lceil C'\rceil$$

We use lemma (B.2) on (IH) and find:

$$\langle\lceil C_1\rceil, \Sigma, \lceil\mathscr{C}\rceil\rangle \twoheadrightarrow \langle\mathbf{0}, \Sigma^\dagger, \lceil\mathscr{C}\rceil\rangle \wedge \langle\lceil C'\rceil, \Sigma^\dagger, \lceil\mathscr{C}\rceil\rangle \Downarrow \Sigma''$$

Which lets us construct the following computation:

$$\begin{aligned}
&\langle\text{if } p.e \text{ then } \lceil C_1\rceil \text{ else } \lceil C_2\rceil;\ \lceil C'\rceil, \Sigma, \lceil\mathscr{C}\rceil\rangle\\
&\to \langle\lceil C_1\rceil;\ \lceil C'\rceil, \Sigma, \lceil\mathscr{C}\rceil\rangle\\
&\twoheadrightarrow \langle\mathbf{0};\ \lceil C'\rceil, \Sigma^\dagger, \lceil\mathscr{C}\rceil\rangle\\
&\to \langle\lceil C'\rceil, \Sigma^\dagger, \lceil\mathscr{C}\rceil\rangle \Downarrow \Sigma''
\end{aligned}$$

– COND-ELSE: This inductive step is symmetric with respect to COND-THEN

– CALL: Inversion gives us:

$$C = X(\vec{p});\ C' \tag{I1}$$
$$X(\vec{q}) = C_X \in \mathscr{C} \tag{I2}$$
$$C^\star = C_X[\vec{q}/\vec{p}] \fatsemi C' \tag{I3}$$
$$\Sigma^\star = \lfloor\Sigma\rfloor \tag{I4}$$

By definition of $\lceil\cdot\rceil$ we have:

$$\lceil C\rceil = X(\vec{p});\ \lceil C'\rceil$$

We use lemma (B.2) on (IH) and find:

$$\langle\lceil C_X[\vec{p}/\vec{q}]\rceil, \Sigma, \lceil\mathscr{C}\rceil\rangle \twoheadrightarrow \langle\mathbf{0}, \Sigma^\dagger, \lceil\mathscr{C}\rceil\rangle \wedge \langle\lceil C'\rceil, \Sigma^\dagger, \lceil\mathscr{C}\rceil\rangle \Downarrow \Sigma''$$

By definition of $\lceil\mathscr{C}\rceil$, then:

$$X(\vec{q}) = \lceil C_X\rceil \in \lceil\mathscr{C}\rceil$$

Which lets us construct the following computation (equivalences on syntactic transformations are proved in Appendix A):

$$\begin{aligned}
&\langle X(\vec{p});\ \lceil C'\rceil, \Sigma, \lceil\mathscr{C}\rceil\rangle\\
&\to \langle\lceil C_X\rceil[\vec{q}/\vec{p}];\ \lceil C'\rceil, \Sigma, \lceil\mathscr{C}\rceil\rangle\\
&= \langle\lceil C_X[\vec{q}/\vec{p}]\rceil;\ \lceil C'\rceil, \Sigma, \lceil\mathscr{C}\rceil\rangle\\
&\twoheadrightarrow \langle\mathbf{0};\ \lceil C'\rceil, \Sigma^\dagger, \lceil\mathscr{C}\rceil\rangle\\
&\to \langle\lceil C'\rceil, \Sigma^\dagger, \lceil\mathscr{C}\rceil\rangle \Downarrow \Sigma''
\end{aligned}$$

□

Putting all the Steps Together

We saw how, starting from a maximal computation

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

We can:

- Replace every non-deterministic aspect of the computation without changing the result, thus letting us reason only on the deterministic subset of the semantics.
- Remove terms which do not affect the computation result, thus letting us ignore them.
- Construct a computation in the instrumented semantics that maintains *low*-equivalence for stores.

Thus, since by definition:

$$\lfloor \lceil \Sigma \rceil^{\Gamma} \rfloor = \Sigma$$

then the *completeness lemma* as stated in (4.9) is proven. □

4.3.2 Completeness of Type Extension

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

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

Proof:
The type judgment is clearly maintained by $\wr \cdot \wr$ because the only effect of the operator is to *remove* instructions. The following is a more general case, easy to verify by inverting the typing rule for sequences (which is basically a *conjunction*):

$$\Delta; \Gamma; \bot \vdash I; C \Rightarrow \Delta; \Gamma; \bot \vdash C$$

We can verify that *lifting* maintains typing by noticing that $\lceil C \rceil$ never introduces a *bracketed* term and for any non-bracketed term the typing rules are syntactically equivalent between **Chor** and **[Chor]**. □

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 executing a step of the semantics gives us a well-typed configuration.* We are now working only in the instrumented semantics, thus, we will omit the redundant brackets when sufficiently clear.
A configuration $\langle C, \Sigma, \mathscr{C} \rangle$ is well-typed for a $pc \in \mathscr{L}$ when:

$$\Delta, \Gamma, pc \vdash C \quad \text{and} \quad \Gamma \vdash \Sigma$$

The preservation lemma is stated as follows:

$$\Delta, \Gamma, pc \vdash C \wedge \Gamma \vdash \Sigma \wedge \langle C, \Sigma, \mathscr{C} \rangle \rightarrow \langle C', \Sigma', \mathscr{C} \rangle \Rightarrow \Delta, \Gamma, pc \vdash C' \wedge \Gamma \vdash \Sigma' \tag{4.13}$$

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

Proof:
The majority of the preservation proof follows directly from [23], thus, we will explain only the part which differs.
The main difference between our instrumented syntax and the language presented in [23] is the presence of recursive procedures.

The inductive case that we have to consider is:

$$\Delta, \Gamma, pc \vdash X(\vec{p}) \tag{H1}$$
$$\langle X(\vec{p}), \Sigma, \mathscr{C} \rangle \rightarrow \langle C_X[\vec{q}/\vec{p}], \Sigma, \mathscr{C} \rangle \tag{H2}$$
$$X(\vec{q}) = C_X \in \mathscr{C} \tag{H3}$$

By inversion on (H1) we find:

$$\Gamma' \in \Delta(X, pc) \tag{H11}$$
$$\Gamma[\vec{q} \mapsto \vec{p}] \equiv_{\{\vec{q}\}} \Gamma' \tag{H12}$$

By definition of Δ, then

$$\Gamma' \in \Delta(X, pc) \Rightarrow \Delta, \Gamma', pc \vdash C_X$$

We now state a lemma which will not be proven in this document (since it is quite long and syntactical), but is part of the *lean* artifact.

$$\begin{aligned} &\Delta, \Gamma', pc \vdash C \Rightarrow \ pn(C) \subseteq \vec{q} \\ &\Rightarrow \ \Gamma'' \equiv_{\{\vec{q}\}} \Gamma' \Rightarrow \ \Delta, \Gamma'', pc \vdash C \end{aligned}$$

From *well-formedness of procedure context* (assumed in 4.4) we know that $pn(C_X) \subseteq \vec{q}$ thus we find

$$\Delta, \Gamma[\vec{q} \mapsto \vec{p}], pc \vdash C_X$$

We now introduce another lemma, proven in the *lean* code but for which the proof (for the same reasons as above) will not be reproduced in this document.

$$\Delta, \Gamma[\vec{q} \mapsto \vec{p}], pc \vdash C \Rightarrow \Delta, \Gamma, pc \vdash C[\vec{q}/\vec{p}]$$

This lemma requires *freshness of formal parameters* to be proven, assumed in 4.1. We now remind ourselves the conclusion to be proven, that is:

$$\Delta, \Gamma, pc \vdash C_X[\vec{q}/\vec{p}] \quad \wedge \quad \Gamma \vdash \Sigma$$

The left-hand side is proven by the previous chain of steps. The right-hand side is easy to prove from hypothesis of the preservation lemma. □

pc subsumption: Subsumption of procedures invocation (used in the proof in [23] for conditional steps) is directly given by the assumption in 4.3

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 Σ_1, Σ_2 in **[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 either exist C_2', Σ_2' s.t.

$$\langle C_2, \Sigma_2, \mathscr{C} \rangle \twoheadrightarrow \langle C_2', \Sigma_2', \mathscr{C} \rangle \wedge C_1' \approx_{low} C_2' \wedge \Sigma_1' \approx_{low} \Sigma_2' \tag{4.14}$$

or that $\langle C_2, \Sigma_2, \mathscr{C} \rangle$ diverges.

The proof of this lemma is mostly unchanged from the one presented in [23] since the differences between the two languages treated are not relevant. For this reason, we omit the proof from this document.

4.4 Main Proof

We remind ourselves the main theorem as stated in (4.2):

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

Proof:
By completeness (4.9) we find:

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

such that

$$\lfloor [\Sigma]'_1 \rfloor = \Sigma'_1 \quad \text{and} \quad \lfloor [\Sigma]'_2 \rfloor = \Sigma'_2 \tag{4.15}$$

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 (4.13) to find

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

That, with (4.15) lets us reduce the proof of $\Sigma'_1 \equiv^\Gamma_{low} \Sigma'_2$ to the proof of $[\Sigma]'_1 \approx_{low} [\Sigma]'_2$

We have now reduced the proof to the following:

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

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 $\langle \cdot, \cdot, \cdot \rangle \Downarrow [\Sigma]'_1$:

- *size 0*: Transition of size zero means that the first choreography is **0**. By definition of $\approx_{low}$ the second is **0** aswell. By hypothesis, the two Σ are equivalent, thus the conclusion follows.

- *size n+1*: This means that the first transition is composed of at least one step of the operational semantics:

$$\langle [C]_1, [\Sigma]_1, \lceil \mathscr{C} \rceil \rangle \rightarrow \langle [C]_1^*, [\Sigma]_1^*, \lceil \mathscr{C} \rceil \rangle$$

  By unwinding (4.14), we find $[C_2]^*$, $[\Sigma]_2^*$ s.t. the low-equivalence is respected (since we are proving *termination insensitive non-interference*, we have as hypothesis that the computation from $\langle [C]_2, [\Sigma]_2, \lceil \mathscr{C} \rceil \rangle$ does not diverge). By preservation (4.13), we find well-typedness and well-formedness needed to use the induction hypothesis, from which we prove the conclusion.

□

Chapter 5

Construction of Δ

We now deal with the creation of the **SecFunCtx** from a *well-formed procedure context* (as defined in 4.4).

Let us remind ourselves the properties which we previously assumed for Δ:

- *Well-typedness with respect to* $\mathscr{C}$: As defined in 4.5, we want

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

  This property can be stated as follows: every declared procedure must type-check under every security environment that Δ admits for it (at any program counter). It is to be noted that the lookup yields an empty set of admissible environments in the case of untypable procedure at a certain pc i.e., $\Delta\ X\ pc = \varnothing$.

- *pc subsumption of context*: For any procedure in Δ, we want the typing to be maintained when lowering the control level, defined formally in 4.3 as:

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

Those are the two properties assumed as true and used for the previous proof of non-interference. In the following sections, we will take ideas and terminology from the practice of *type reconstruction* [26].

5.1 Context Reconstruction Algorithm

The key idea behind constructing Δ is inferring from every function $X(\vec{q}) = C_X$ in $\mathscr{C}$ a list of *constraints* such that, if Γ satisfies those constraints, then it well-types C_X.

5.1.1 Local Expression Reconstruction

Let us start dealing with local expressions. The idea here is to infer a symbolic *bound* which can delay the computation of the security level of an expression. We define the following syntax:

$$\psi ::= \bot \mid x \mid \psi \sqcup \psi$$

Where $\bot$ is an arbitrary symbol and x is taken from the previously defined set **Var**. We define **ExprBound** as the set of the objects generated by the nonterminal ψ. We associate a *semantics* to the syntactic category **ExprBound**:

$$[\![\psi]\!] : \mathbf{SecPLab} \to \mathscr{L}$$

That is, the semantics of a bound is defined as a map that computes the level associated with a local expression given an security level assignment for the variables present. We define it recursively on the structure of ψ:

$$\begin{aligned} [\![\bot]\!]\ \gamma &= \bot \\ [\![x]\!]\ \gamma &= \gamma\ x \\ [\![\psi_1 \sqcup \psi_2]\!]\ \gamma &= ([\![\psi_1]\!]\ \gamma) \sqcup ([\![\psi_2]\!]\ \gamma) \end{aligned}$$

We now define the procedure of bound reconstruction $\vdash e \rhd \psi$ that, given an **Expr** e, returns the associated bound:

$$\vdash v \rhd \bot \qquad \vdash x \rhd x \qquad \frac{\sigma \vdash e_1 \rhd \psi_1 \quad \cdots \quad \sigma \vdash e_n \rhd \psi_n}{\sigma \vdash f(e_1, \ldots, e_n) \rhd \sqcup_{i=1}^{n} \psi_i}$$

We now state and prove the *correctness* of this algorithm with respect to the typing relation for **Expr** previously defined. Formally:

$$\vdash e \rhd \psi \ \wedge\ [\![\psi]\!]\ \gamma = \ell \implies \gamma \vdash e : \ell \tag{5.1}$$

Proof:
We proceed by induction on e

- *constant:* we have the following:

  $$e = v \quad \psi = \bot \quad \ell = \bot$$

  The conclusion comes easily from the corresponding typing rule in 3.3.2.

- *variable:* we have the following:

  $$e = x \quad \psi = \gamma\ x \quad \ell = \gamma\ x$$

  The conclusion comes easily from the corresponding typing rule in 3.3.2.

- *local function:* We have the following:

$$e = f(e_1, \ldots, e_n) \quad \psi = \sqcup_{i=1}^{n} \psi_i \quad \ell = [\![\sqcup_{i=1}^{n} \psi_i]\!]\ \gamma \tag{H1}$$
$$[\![\psi_i]\!] = \ell_i \Rightarrow \gamma \vdash e_i : \ell_i \tag{IH}$$

  By definition of $[\![\cdot]\!]$ and (H1) we find:

$$\ell_i = [\![\psi_i]\!]\ \gamma \qquad \ell = \sqcup_{i=1}^{n} \ell_i$$

  Which can be used with (IH) to construct the typing derivation for:

$$\gamma \vdash f(e_1, \ldots, e_n) : \ell$$

□

5.1.2 Choreography Reconstruction

The main idea remains the same as before. We firstly define the syntactic category **Bound**:

$$\Psi ::= \bot \mid p.x \mid \eta \mid \Psi \sqcup \Psi$$

where $p.x \in \mathbf{Pid} \times \mathbf{Var}$, η taken from a set **Fresh** of variables *always considered sufficiently fresh*. η will represent the security level of *pc* in bounds.
We define the syntactic category **Constraint** (the name is self-explanatory):

$$\omega ::= \Psi \sqsubseteq p.x$$

We define a *constraint context* (**ProcConstr**) δ : **ProcName** $\rightarrow$ **Finset Constraint** which maps every procedure to the associated finite set of constraints.
We can now define the constraint reconstruction procedure $\delta, \eta \vdash C \rhd \Psi$ as follows:

$$\frac{\vdash e \rhd \psi_e \quad \Psi_p = \text{bind } \psi_e\ p}{\delta, \eta \vdash p.x := e \rhd \{\Psi_p \sqsubseteq p.x, \eta \sqsubseteq p.x\}}$$

$$\frac{\vdash e \rhd \psi_e \quad \Psi_p = \text{bind } \psi_e\ p}{\delta, \eta \vdash p.e \rightarrow q.x \rhd \{\Psi_p \sqsubseteq q.x, \eta \sqsubseteq q.x\}}$$

$$\frac{\vdash e \rhd \psi_e \quad \Psi_p = \text{bind } \psi_e\ p \quad \delta, \eta_1' \vdash C_1 \rhd E_1 \quad \delta, \eta_2' \vdash C_2 \rhd E_2}{\delta, \eta \vdash \text{if } p.e \text{ then } C_1 \text{ else } C_2 \rhd E_1[\eta_1'/\eta \sqcup \Psi_p] \cup E_2[\eta_2'/\eta \sqcup \Psi_p]}$$

$$\frac{\delta\ X = \vec{q}, \eta_X, E_X}{\delta, \eta \vdash X(\vec{p}) \rhd E_X[\vec{q}/\vec{p}][\eta_X/\eta]}$$

$$\frac{\delta, \eta \vdash I \rhd E_I \quad \delta, \eta \vdash C \rhd E_C}{\delta, \eta \vdash I; C \rhd E_I \cup E_C}$$

$$\delta, \eta \vdash \mathbf{0} \rhd \emptyset$$

Substitution in a constraint set is defined naturally.
The function bind is defined as *lifting* an **ExprBound** into a **Bound** by binding every **Var** into the corresponding **Pid** $\times$ **Var**

We can already state a first lemma, which will be useful in the following:
Single eta: For each constraint reconstruction E such that:

$$\delta, \eta \vdash C \rhd E$$

Then, the only element of **Fresh** that can appear in E is η, given that this property is respected for every η_X, E_X in δ. It is to be noted that it is admitted for E to have no element of **Fresh**.
Proof: The proof easily follows by induction. □

We now define a couple of functions that relate constraints with security contexts:

$$\underline{\text{cansolve}}\ (E : \mathbf{Finset\ Constraint})\ (\Gamma : \mathbf{SecCLab})\ (pc : \mathscr{L}) : \mathbf{Proc} :=$$
$$\forall\ (\Psi \sqsubseteq p.x) \in E,\ [\![\Psi]\!]\ \Gamma\ pc \sqsubseteq \Gamma\ p.x$$

Where $[\![\cdot]\!]$ for **Bound** is the natural extension from the one for **ExprBound**.

$$\underline{\text{gen}}\ (\delta : \mathbf{ProcConstr}) : \mathbf{ProcName} \to \mathscr{L} \to \mathbf{Finset\ SecCLab} :=$$
$$\lambda\ X.\ \lambda\ pc.\ \{\Gamma \mid \underline{\text{cansolve}}\ (\delta\ X)\ \Gamma\ pc\}$$

5.2 Proof of Well-Typedness

We are looking to prove that given a $\mathscr{C}$ we can use the type reconstruction algorithm to create a δ such that $\underline{\text{gen}}\ \delta$ creates a *well-typed context* (4.5) with respect to $\mathscr{C}$.

5.2.1 Creating δ

We start by defining a monotonic operator $\phi_{\mathscr{C}} : \mathbf{ProcConstr} \to \mathbf{ProcConstr}$ as follows:

$$\phi_{\mathscr{C}} : \delta \mapsto (X \mapsto E_X)$$

Where $X(\vec{q}) = C_X \in \mathscr{C}$ and E_X is computed as $\delta, \eta \vdash C_X \rhd E_X$
$\phi_{\mathscr{C}}$ creates a delta by applying *one pass* of constraint reconstruction to every procedure in the procedure context. This is needed because procedures can be mutually recursive, thus, we need multiple passes to construct all constraints correctly. Proof of monotonicity of $\phi_{\mathscr{C}}$ is addressed in Appendix C.1.

5.2.2 One step soundness

We proceed by stating and proving the following:

$$\begin{aligned} &\Delta = \underline{\text{gen}}\ \delta \\ &\Rightarrow \delta, \eta \vdash C \rhd E \\ &\Rightarrow \underline{\text{cansolve}}\ E\ \Gamma\ pc \\ &\Rightarrow \Delta, \Gamma, pc \vdash C \end{aligned} \tag{5.2}$$

Proof: We proceed by induction on C

- *case: assignment.* By inversion and unrolling the definition of $\underline{\text{cansolve}}$ we have the following hypothesis:

$$\vdash e \rhd \psi_e \tag{H1}$$
$$\Psi_p = \text{ bind } \psi_e\ p \tag{H2}$$
$$[\![\Psi_p]\!]\ \Gamma\ pc \sqsubseteq \Gamma\ p.x \tag{H3}$$
$$[\![\eta]\!]\ \Gamma\ pc \sqsubseteq \Gamma\ p.x \tag{H4}$$

  We start by constructing:

$$\ell = [\![\psi_e]\!]\ (\Gamma\ p)$$

  By hypothesis (H1) and lemma (5.1) we find:

$$\Gamma\ p \vdash e : \ell$$

  By (H2) and definition of $[\![\cdot]\!]$ we find:

$$\ell = [\![\Psi_p]\!]\ \Gamma\ pc \qquad pc = [\![\eta]\!]\ \Gamma\ pc$$

  By (H3) and (H4) we thus find the premise necessary for typing

$$\Delta, \Gamma, pc \vdash p.x := e$$

- *case: communication.* The case follows similarly from the previous one

- *case: conditional.* We have the following hypothesis:

$$\vdash e \rhd \psi_e \tag{H1}$$
$$\Psi_p = \text{ bind } \psi_e\ p \tag{H2}$$
$$\delta, \eta_1' \vdash C_1 \rhd E_1 \tag{H3}$$
$$\delta, \eta_2' \vdash C_2 \rhd E_2 \tag{H4}$$
$$\underline{\text{cansolve}}\ (E_1[\eta/\eta_1' \sqcup \Psi_p]\ \cup\ E_2[\eta/\eta_2' \sqcup \Psi_p])\ \Gamma\ pc \tag{H5}$$
$$\forall\ pc^\star, \underline{\text{cansolve}}\ E_1\ \Gamma\ pc^\star \Rightarrow \Delta, \Gamma, pc^\star \vdash C_1 \tag{IH1}$$
$$\forall\ pc^\star, \underline{\text{cansolve}}\ E_2\ \Gamma\ pc^\star \Rightarrow \Delta, \Gamma, pc^\star \vdash C_2 \tag{IH2}$$

As before, we can use (H1), lemma (5.1) and (H2) to find:

$$\ell = [\![\psi_e]\!]\ (\Gamma\ p)\ \ \text{(A1)} \qquad \Gamma\ p \vdash e : \ell \qquad \ell = [\![\Psi_p]\!]\ \Gamma\ pc$$

We will now concentrate ourself to the typing of the *then* branch, since the other one follows by symmetry.
By reasoning on the definition of $\underline{\text{cansolve}}$ and (H5) we find:

$$\underline{\text{cansolve}}\ E_1[\eta/\eta_1' \sqcup \Psi_p]\ \Gamma\ pc$$

We now apply the following lemma (proved in Appendix C.2)

$$\begin{aligned}&\underline{\text{cansolve}}\ E[\eta/\eta' \sqcup \Psi]\ \Gamma\ pc \quad \wedge \quad (\forall pc', [\![\Psi]\!]\ \Gamma\ pc' = \ell')\\ &\Longrightarrow \underline{\text{cansolve}}\ E\ \Gamma\ (pc \sqcup \ell')\end{aligned}$$

We see how the second premise is true for the considered ℓ: no pc appears in (A1).
We thus use (IH1) to find the required typing:

$$\Delta, \Gamma, (pc \sqcup \ell) \vdash C_1$$

- *case: procedure call.* We have the following hypothesis:

$$\Delta = \underline{\text{gen}}\ \delta \qquad \text{(H1)}$$
$$\delta\ X = \vec{q}, \eta_X, E_X \qquad \text{(H2)}$$
$$\underline{\text{cansolve}}\ E_X[\vec{q}/\vec{p}\,][\eta_X/\eta]\ \Gamma\ pc \qquad \text{(H3)}$$

We introduce the following lemmas: the first one follows from *Single eta* and the second one is proven in Appendix C.3:

$$\begin{aligned}&\underline{\text{cansolve}}\ E[\eta/\eta']\ \Gamma\ pc \Rightarrow \underline{\text{cansolve}}\ E\ \Gamma\ pc\\ &\underline{\text{cansolve}}\ E[\vec{q}/\vec{p}\,]\ \Gamma\ pc \Rightarrow \underline{\text{cansolve}}\ E\ \Gamma[\vec{q} \mapsto \vec{p}\,]\ pc\end{aligned}$$

We apply these lemmas to (H3), which gives us:

$$\underline{\text{cansolve}}\ E_X\ \Gamma[\vec{q} \mapsto \vec{p}\,]\ pc$$

From (H2), (H1) and the definition of $\underline{\text{gen}}$ we get:

$$\Gamma[\vec{q} \mapsto \vec{p}\,] \in \Delta\ X\ pc$$

Thus we can use the typing rule for procedure calls, selecting $\Gamma' = \Gamma[\vec{q} \mapsto \vec{p}]$. By reflexivity, is it trivial to show:

$$\Gamma[\vec{q} \mapsto \vec{p}] \equiv_{\{\vec{q}\}} \Gamma[\vec{q} \mapsto \vec{p}]$$

□

5.2.3 Well-Typed δ

By our definition, **ProcConstr** can be considered as finite, this holds because we are working with a finite $\mathscr{L}$ and because the number of procedure names mentioned in a finite choreography is always finite. By Knaster-Tarski theorem [30], there exists a *least fixed point* for $\phi_{\mathscr{C}}$. Let us now reason on the previous results:
A trivial lemma of 5.2 is the following:

$$\begin{aligned}&\Delta = \underline{\text{gen}}\ \delta_1\\ &\Rightarrow \delta_2 = \phi_{\mathscr{C}}\ \delta_1\\ &\Rightarrow \underline{\text{cansolve}}\ (\delta_2\ X)\ \Gamma\ pc\\ &\Rightarrow \Delta, \Gamma, pc \vdash C_X\end{aligned}$$

We can thus see that, choosing as δ_1, a fixed point $\mu\phi_{\mathscr{C}}$, and by using the definition of $\underline{\text{gen}}$, this lemma becomes:

$$\begin{aligned}&\Delta = \underline{\text{gen}}\ \mu\phi_{\mathscr{C}}\\ &\Rightarrow \Gamma\ \in\ \Delta\ X\ pc\\ &\Rightarrow \Delta, \Gamma, pc \vdash C_X\end{aligned}$$

Which makes Δ a *well-typed security procedure context.* □

5.3 Proof of pc Subsumption

We remind ourselves the statement of pc subsumption for contexts:

$$\Delta = \underline{\text{gen}}\ \delta\ \wedge\ \Gamma \in \Delta\ X\ pc\ \wedge\ pc' \sqsubseteq pc \Longrightarrow \Gamma \in \Delta\ X\ pc'$$

Proof: By definition of $\underline{\text{gen}}$ we can rewrite the statement as follows:

$$\underline{\text{cansolve}}\ E\ \Gamma\ pc\ \wedge\ pc' \sqsubseteq pc\ \Longrightarrow \underline{\text{cansolve}}\ E\ \Gamma\ pc'$$

Which we can further decompose by unfolding the definition of $\underline{\text{cansolve}}$ into:

$$[\![\Psi]\!]\ \Gamma\ pc \sqsubseteq p.x \wedge\ pc' \sqsubseteq pc \Longrightarrow [\![\Psi]\!]\ \Gamma\ pc' \sqsubseteq p.x$$

Which becomes:

$$pc' \sqsubseteq pc \Longrightarrow [\![\Psi]\!]\ \Gamma\ pc' \sqsubseteq [\![\Psi]\!]\ \Gamma\ pc$$

Which is easily proven reasoning inductively on the structure of Ψ. □

5.4 Termination

Since all the operations functions defined work on finite structures, there will always be a terminating algorithm to compute them.

5.5 Extension to Full Type Inference

The algorithm we defined is very similar to type-inference [26], but a few aspects need to be treated formally to consider it so. Let us now give an overview of the main ones.

- *Reconstruction of constraints for the configuration's choreography.* This should not be any harder than to reconstruct constraints from the procedure context. A way forward would be to consider the configuration's choreography as an *unnamed* procedure inside the context.
- *Proof of completeness of the constraint reconstruction algorithm with respect to the type checking.* This requires creating an order relation on the **Constraint** set, such that it is possible to define a *minimal* set of constraints. Afterwards, the proof would follow by proving that every Γ typing a choreography would satisfy the minimal set of constraints reconstructed for that choreography.

We do not address these topics in this thesis because of the limited time available.

Chapter 6

Lean mechanization

6.1 Naming Conventions

In the following, a few naming conventions are different and should be considered:

$$\texttt{L} \triangleq \mathscr{L} \qquad \texttt{ps} \triangleq \vec{p}$$
$$\texttt{D} \triangleq \Delta \qquad \texttt{G} \triangleq \Gamma$$
$$\texttt{BrChor} \triangleq [\textbf{Chor}]$$

6.2 Imported Definitions and Results

6.2.1 Encoding of $\mathscr{L}$

Mathlib 4 [20] was used as library in the project. This was done so that we could take advantage of the already mechanized math results.
The biggest contribution taken from the library is the `Mathlib.Order.Lattice` module, which was used to encode $\mathscr{L}$ by defining it as a *type variable* which implements the `Lattice`, `DecidableLE` and `Bot` *type classes*.

6.2.2 Reference choreographies

Ongoing research is being done by Xueying Qin and Fabrizio Montesi at SDU to mechanize the language defined in 2.1. My work builds on theirs, importing the needed definitions and results. The main imported constructs are the following: procedure context implementation, syntax and semantics for reference choreographies.

6.3 Project Structure

The project is divided in multiple files to deal with the length and complexity of the proof. These files are the following:

- `Common.lean` This file contains all the definitions which are needed by all the other files in the project. For example *security labeling* 3.1:

  ```
  -- Security Labeling for Variables of a Processes
  abbrev SecPLab := Var -> L
  -- Security Labeling for Variables "Globally"
  abbrev SecCLab := Pid -> @SecPLab L
  ```

  Or *security context for procedures*:

  ```
  def SecFunCtx :=
      ProcName -> L -> (List Pid x Finset (@SecCLab L))
  ```

- `Syntax.lean` This file contains the definition of the instrumented syntax, encoded as an inductive type, and all related lemmas and definition (for example *substitution*, *low equality*).

- `Semantics.lean` This file contains both the natural semantics as defined in 4.1 and the operational semantics for instrumented choreographies. Both relations are defined as inductive types. Let us see the signatures for them:

  ```
  inductive Chor.natsem {sig: EvalSig}:
      CConf -> List TransitionLabel -> CStore -> Prop

  inductive BrChor.lto {G: @SecCLab L}:
      BrConf -> BrConf -> Prop
  ```

- `Typing.lean` This file contains the typing relation as defined in 3.3 and all the helper lemmas used in the proofs. The relation is encoded as an mutually inductive type, with the following signature:

  ```
  mutual
  inductive tj_i {D: @SecFunCtx L} {G: @SecCLab L}:
      L -> Instruction -> Prop

  inductive tj_c {D: @SecFunCtx L} {G: @SecCLab L}:
      L -> Choreography -> Prop
  end
  ```

- `LowerLiftStx.lean` In this file are defined and mechanized all the lowering/lifting functions which this document treated at a more abstract level. A great deal of helper and inversion lemmas are needed to make the definitions usable in the proofs. To cite some numbers, the file is composed of 1389 lines of code for 8 lowering/lifting functions. Let us see a few cases of the lifting function for choreography, to see how the issue of $\wr \cdot \wr$ was dealt with:

```
mutual
def lift_i (i: Instruction) (H1: no_choice_i i)
    (H2: no_rtcall_i i): BrChor := match i with
    | Instruction.assign p e x => BrChor.ass p e x
    | Instruction.cond p e c1 c2 => BrChor.cond p e
        (lift_c c1 (...) (...))
        (lift_c c2 (...) (by dsimp [no_rtcall_i] at H2;
                            apply And.right H2))
    | Instruction.rtcall q x ps c => (by exfalso; apply H2)

def lift_c (c: Choreography) (H1: no_choice_c c)
    (H2: no_rtcall_c c): BrChor := match c with
    | Choreography.nil => BrChor.nil
    | Choreography.seq i c' => BrChor.seq
        (lift_i i (...) (...))
        (lift_c c' (...) (by dsimp [no_rtcall_c] at H2;
                            apply And.right H2))
end
```

  Where `no_rtcall` is defined as follows (in file `Common.lean`):

```
mutual
def no_rtcall_i (i: Instruction): Prop := match i with
    | Instruction.rtcall _ _ _ _ => False
    | Instruction.cond _ _ c1 c2 => no_rtcall_c c1 /\ no_rtcall_c c2
    | _ => True

def no_rtcall_c (c: Choreography): Prop := match c with
    | Choreography.nil => True
    | Choreography.seq i c' => no_rtcall_i i /\ no_rtcall_c c'
end
```

- `Completeness.lean` The main theorem defined in this file is 4.9. I was not able to finish the proof in lean for time related reasons. Since the completeness proof is not mechanized, it was described in deeper detail in this document.

- `Preservation.lean` The main theorem proved in this file is 4.13. In the same file are defined and proved helper lemmas (for example *pc subsumption* and typing for instrumented local expressions).
- `Unwinding.lean` The main theorem proved in this file is 4.14. In the same file are defined and proved helper lemmas (for example the *high step lemma* [23] and helper lemmas for low equality of instrumented stores).
- `NonInt.lean` This file contains the main proof, corresponding to 4.2. All the other files of the project are imported in this one to be able to define and prove the main theorem.

6.4 Coverage of the Mechanization

Many results described in this document have been mechanized and are present in the lean code, but not all. Two main results are admitted and are proved only by pen and paper: *completeness of the reference semantics with respect to the instrumented semantics* and *construction of* Δ.
These two sections were not represented in lean for time constraints: any non-trivial lemma requires me to have first a pen and paper presentation. This is because while writing lean code it is very easy to concentrate on the syntactical aspects of the proof and to lose sight of the overall strategy. Moreover, once the proof strategy is clear, the process of encoding it in lean can be very time consuming, and it is sometimes difficult to correctly predict which proof steps are going to be the most challenging. Sometimes half a page paper presentations become hundreds and hundreds of lines of code.

6.5 An Illustrative Example

We will now show the lean implementation of the following lemma, used in section 4.3.3. Let us remind ourselves the lemma statement:

$$\Delta, \Gamma[\vec{q} \mapsto \vec{p}\,], pc \vdash C \Longrightarrow \Delta, \Gamma, pc \vdash C[\vec{q}/\vec{p}\,]$$

In lean, the same statement is expressed as:

```
def tj_substs {D: @SecFunCtx L} {G: SecCLab} {pc: L} {c: BrChor}
    {ps qs: List Pid} {Hlen: ps.length = qs.length}:
    fresh_functx D
    -> @tj_bc _ _ low tjeval D (secclab_substs G ps qs Hlen) pc c
    -> @tj_bc _ _ low tjeval D G pc (c.substitutions ps qs Hlen)
```

As we can see, most of the assumptions which are treated as implicit in the pen and

paper proof here need to be made explicit.

- *Same length of $\vec{p}$ and $\vec{q}$*: In this document, we always assumed same lenght from the arguments of every substitution operation. Let us now see how the substitution is defined in the lean code:

```
def BrChor.substitutions (c: BrChor) (ps qs: List Pid)
    (H: ps.length = qs.length): BrChor :=
        match ps, qs with
        | hp::tp, hq::tq =>
            BrChor.substitutions (c.substitution hp hq) tp tq
                (by simp_all)
        | [], [] => c
```

  Where `BrChor.substitutions` performs **Pid** substitution for choreographies as defined in 2.1.3. We see how the function is only defined for inputs of the same length, and how this requirement allows us to ignore the impossible pattern matching cases.

- *Fresh parameters for Δ*: We want to encode the assumption *freshness of formal parameters* as defined in 4.1.

```
def fresh_functx (D: @SecFunCtx L) :=
    forall (pc: L) (X: ProcName) (qs: List Pid) (ps: Finset Pid),
    qs = Prod.fst (D X pc) -> Disjoint ps qs.toFinset
```

  It is to be noted how this property is too restricting, since it can be used with $qs = ps$ to prove that every procedure has an empty list of arguments. While this is technically incorrect, it is a necessary requirements for the semantics as specified in 6.2 (which were thus outside of my control). A cleaner solution would be to define $\mathscr{C}$ as a *partial* function with a finite image, such that we can enforce a partitioning of the set **Pid** between the procedures and the *main* choreography.

Context Renaming

We now see how the function `secclab_substs` encodes the *context renaming* operator as defined in 3.1. The function is defined as follows:

```
def secclab_substs (G: @SecCLab L) (p q: List Pid)
    (h: p.length = q.length): @SecCLab L := match p, q with
       | hp::tp, hq::tq =>
            secclab_subst (secclab_substs G tp tq
                (by simp only
                    [List.length_cons, add_left_inj] at h
                 apply h))
            hp hq
       | [], [] => G
```

Where `secclab_subst` is the function dealing with scalar substitution, defined as follows:

```
def secclab_subst (G: @SecCLab L) (p q: Pid): @SecCLab L :=
    lambda r => if p = r then G q else G r
```

Proof

For the rest of the chapter there is an important note to make: my proof style in lean can be considered mostly *backward chaining* [1], that is, I tend to manipulate the goal more than the hypothesis. Let us now see the proof body in lean, omitting the uninteresting induction cases.

```
intro Hfresh H1
induction H1
case tass l' Htjeval Hsub =>
    rw [BrChor.substitutions.inversion_ass]
    apply tj_bc.tass (l' := l')
    case a => simp_all [sec_substs_of_pid_substs]
    case a => simp_all [sec_substs_of_pid_substs]
case tcond Heval _ _ IH1 IH2 =>
    rw [BrChor.substitutions.inversion_cond]
    rw [sec_substs_of_pid_substs (G := G)] at Heval
    apply tj_bc.tcond Heval IH1 IH2
case tcall G' pc x ps' qs' Hlen Hg Hqs Hgam =>
    rw [BrChor.substitutions.inversion_call]
    dsimp [fresh_functx] at Hfresh
    apply tj_bc.tcall Hg Hqs
    have Hdis1 := Disjoint.symm
        (Hfresh pc x qs' ps'.toFinset Hqs)
    have Hdis3 := Disjoint.symm (Hfresh pc x qs' qs.toFinset Hqs)
    apply gam_res_eq_trans
    apply (gam_res_eq_symm (gamsubsts_helper Hdis1 Hdis3))
    apply Hlen
    apply Eq.trans Hlen pids_substs_length
    apply Hgam
```

Let us unpack what happens in this code snippet: The proof follows by induction on the typing derivation `tj_bc`. We will explain the cases for *assignment*, *conditionals* and *procedure calls*.

- *assignment*: The inductive cases gives us hypothesis for the typing of the expression (`Htjeval`) and on the order relation between ℓ and $\Gamma\ p\ x$ (`Hsub`). The first step rewrites the goal from:
  `tj_bc D G pc ((BrChor.ass p e x).substitutions ps qs Hlen)`
  to `tj_bc D G pc (BrChor.ass (pid_substs p ps qs Hlen) e x)`. Thus enabling us to use the typing rule for assignments. The two assumptions of the typing rule are discharged by the proof search mechanism of lean, using the

following helper lemma (which is used on `Heval` to make it compatible with the needed assumption):

```
lemma sec_substs_of_pid_substs {G: @SecCLab L}
    {r: Pid} {ps qs: List Pid} {Hlen}:
    (secclab_substs G ps qs Hlen) r = G (pid_substs r ps qs Hlen)
```

- *conditional*: The inductive case gives us hypothesis for the typing of the expression (`Heval`), and two inductive hypothesis for the two branches (`IH1`, `IH2`). The typing of the expression is dealt exactly as in the previous case, but to be able to use the typing rule on the goal we need to apply the inductive hypothesis.

- *procedure call*: We first, as we did in the previous inductive cases, invert the process substitution to be able to use the call typing rule. The function lookup part of the typing derivation remains the same (which finds $\Gamma', \vec{r}$), thus it remains to prove the following:

$$\Gamma' \equiv_L \Gamma[\vec{q} \mapsto \vec{p}\,][\vec{r} \mapsto \vec{s}] \Longrightarrow \Gamma' \equiv_L \Gamma[\vec{q} \mapsto \vec{p}\,[\vec{r}/\vec{s}]]$$

From the freshness of the procedure parameters we have that

$$\vec{r} \cap (\vec{q} \cup \vec{p}\,) = \emptyset$$

And from the fact that $\equiv_L$ is an equivalence relation, we can reduce the goal to

$$\begin{aligned} &\vec{r} \cap \vec{q} = \emptyset \\ &\Longrightarrow \vec{r} \cap \vec{p} = \emptyset \\ &\Longrightarrow \Gamma[\vec{q} \mapsto \vec{p}\,][\vec{r} \mapsto \vec{s}] \equiv_L \Gamma[\vec{q} \mapsto \vec{p}\,[\vec{r}/\vec{s}]] \end{aligned}$$

Which is proven by the lemma `gamsubsts_helper`, as can be seen by it's signature:

```
lemma gamsubsts_helper {ps qs rs ss: List Pid}
    {G: @SecCLab L} {Hlen1 Hlen2 Hlen3}:
    Disjoint ss.toFinset rs.toFinset
    -> Disjoint ss.toFinset qs.toFinset
    -> gam_res_eq ss.toFinset
        (secclab_substs
            (secclab_substs G ps qs Hlen1) ss rs Hlen2)
        (secclab_substs G ss (pids_substs rs ps qs Hlen1) Hlen3)
```

The proof of this lemma will be described in the next section.

□

Proof of `gamsubsts_helper`

To understand it, we need to detail two more definitions:

```
def gam_res_eq (ps: Finset Pid) (g1 g2: @SecCLab L): Prop :=
    forall (p: Pid), p in ps -> g1 p = g2 p

def Pid.substitution (p r s: Pid): Pid :=
  if p = r then s else p

def pid_substs (r: Pid) (ps qs: List Pid)
    (H: ps.length = qs.length): Pid :=
    match ps, qs with
    | hp::tp, hq::tq =>
        pid_substs (r.substitution hp hq) tp tq (by simp_all)
    | [], [] => r

def pids_substs (rs ps qs: List Pid)
    (H: ps.length = qs.length): List Pid :=
    match rs with
    | h::t => (pid_substs h ps qs H) :: (pids_substs t ps qs H)
    | [] => []
```

Which are considered self-explanatory.
We also describe two helper lemmas:

```
lemma helper23718 {G G': @SecCLab L} {qs ps ps': List Pid}
    (Hlen1: qs.length = ps.length)
    (Hlen2: qs.length = ps'.length):
    Disjoint qs.toFinset ps.toFinset
    -> Disjoint qs.toFinset ps'.toFinset
    -> List.Forall (lambda a => G (Prod.fst a) = G' (Prod.snd a))
                   (List.zip ps ps')
    -> gam_res_eq qs.toFinset (secclab_substs G qs ps Hlen1)
        (secclab_substs G' qs ps' Hlen2)
```

This lemma says that, under correct disjointness conditions, parallel substitutions of security labels are determined purely by the *positionwise* agreement of the labels being plugged in.

```
lemma helper7489376 {fst snd ps qs Hlen} {rs: List Pid}:
    (fst, snd) in (rs.zip (pids_substs rs ps qs Hlen))
    -> (fst in rs) /\ snd = pid_substs fst ps qs Hlen
```

This lemma captures the intended *position-preserving* view of substitutions on pid

lists.

Now that we have all the necessary background, we can look at the body of the proof:

```
intro Hdis1 Hdis2
have Hdis3: Disjoint ss.toFinset
    (pids_substs rs ps qs Hlen1).toFinset
        := disjoint_of_substs2 Hdis1 Hdis2
apply helper23718 _ _ Hdis1 Hdis3
rw [List.forall_iff_forall_mem]
intro a Ha
cases a
case mk fst snd =>
simp
rw [sec_substs_of_pid_substs]
have A1 := helper7489376 Ha
rw [And.right A1]
```

And see that basically the conclusion follows from two main ideas. The first one is that disjointness of the process names lets us reason independently on every element of $\vec{q}$. The second one is triangular reasoning which proves the equation in every possible case.

Appendices

Appendix A

Proofs on Syntactic Transformations

A.1 Sequential Composition and $\wr \cdot \wr$

We need to prove the following:

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

Proof: This proof follows by induction on C_1, unrolling the definitions of the operators involved.

A.2 Process Substitution and $\wr \cdot \wr$

We need to prove the following:

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

Proof: This proof follows by induction on C, unrolling the definitions of the operators involved.

A.3 Store Update and $\lfloor \cdot \rfloor$

We need to prove the following:

$$\lfloor \Sigma \rfloor [p.x \mapsto v] = \lfloor \Sigma [p.x \mapsto [v]] \rfloor$$

Proof: We defined equivalence between stores as extensional equivalence between maps. This theorem follows from triangular reasoning:

For $q.y$ different from $p.x$:

$$\lfloor \Sigma \rfloor [p.x \mapsto v]\ q.y = \lfloor \Sigma \rfloor\ q.y$$
$$\lfloor \Sigma [p.x \mapsto [v]] \rfloor\ q.y = \lfloor \Sigma \rfloor\ q.y$$

While simultaneously:

$$\lfloor \Sigma \rfloor [p.x \mapsto v]\ p.x = v$$
$$\lfloor \Sigma [p.x \mapsto [v]] \rfloor\ p.x = \lfloor [v] \rfloor = v$$

A.4 Process Substitution and $\lceil \cdot \rceil$

We need to prove the following:

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

Proof: This proof follows by induction on C, unrolling the definitions of the operators involved.

Appendix B

Decomposition of Sequential Composition Execution

B.1 [Chor] Sequential Composition

We start by proving the following lemma:

$$\begin{array}{l} \langle C_1; C_2, \Sigma, \mathscr{C}\rangle \Downarrow \Sigma'' \\ \Rightarrow \exists \Sigma', \langle C_1, \Sigma, \mathscr{C}\rangle \twoheadrightarrow \langle \mathbf{0}, \Sigma', \mathscr{C}\rangle \quad \wedge \quad \langle C_2, \Sigma', \mathscr{C}\rangle \Downarrow \Sigma'' \end{array} \tag{B.1}$$

Proof: We proceed by induction on $\cdot \Downarrow \cdot$

- *case nil:* This case is not possible because there are no C_1, C_2 such that $\mathbf{0} = C_1; C_2$. Thus the goal follows by *ex-falso quodlibet*.
- *case step:* the inductive case gives us the following hypothesis:

$$\langle C_1; C_2, \Sigma, \mathscr{C}\rangle \to \langle C^\star, \Sigma^\star, \mathscr{C}\rangle \tag{H1}$$

$$\begin{array}{l} \forall C_1^\star\ C_2^\star, \quad C^\star = C_1^\star; C_2^\star \\ \Rightarrow \exists \Sigma^\heartsuit,\ \langle C_1^\star, \Sigma^\star, \mathscr{C}\rangle \to \langle \mathbf{0}, \Sigma^\heartsuit, \mathscr{C}\rangle \ \wedge\ \langle C_2^\star, \Sigma^\heartsuit, \mathscr{C}\rangle \Downarrow \Sigma'' \end{array} \tag{IH}$$

  By inversion on (H1) we find two possible cases:

  - $\langle C_1, \Sigma, \mathscr{C}\rangle \to \langle C_1^\star, \Sigma^\star, \mathscr{C}\rangle$ with $C^\star = C_1^\star; C_2$
  - $\langle \mathbf{0}; C_2, \Sigma, \mathscr{C}\rangle \to \langle C_2, \Sigma^\star, \mathscr{C}\rangle$ thus with $C^\star = C_2$

  Both cases follow easily from the previous hypothesis. □

B.2 Chor Sequential Composition

We finally prove the following statement:

$$\begin{array}{l}\langle\lceil C_1 \fatsemi C_2\rceil, \Sigma, \mathscr{C}\rangle \Downarrow \Sigma'' \\ \Rightarrow \exists \Sigma', \langle\lceil C_1\rceil, \Sigma, \mathscr{C}\rangle \twoheadrightarrow \langle \mathbf{0}, \Sigma', \mathscr{C}\rangle \quad \wedge \quad \langle\lceil C_2\rceil, \Sigma', \mathscr{C}\rangle \Downarrow \Sigma''\end{array} \tag{B.2}$$

Proof: We proceed by induction on C_1:

- *case* $\mathbf{0}$*:* by definition of $\cdot \fatsemi \cdot$ we have that:

  $$\lceil C_1 \fatsemi C_2\rceil = \lceil C_2\rceil$$

  And the conclusion follows by reflexivity of $\cdot \twoheadrightarrow \cdot$, with $\Sigma' = \Sigma$

- *case* $I_1; C_1'$*:* As *induction hypothesis* we have that, for all $\Sigma^\heartsuit$

  $$\langle\lceil C_1' \fatsemi C_2\rceil, \Sigma^\heartsuit, \mathscr{C}\rangle \Downarrow \Sigma'' \Rightarrow \exists \Sigma^\star, \langle\lceil C_1'\rceil, \Sigma^\heartsuit, \mathscr{C}\rangle \twoheadrightarrow \langle \mathbf{0}, \Sigma^\star, \mathscr{C}\rangle \quad \wedge \quad \langle\lceil C_2\rceil, \Sigma^\star, \mathscr{C}\rangle \Downarrow \Sigma''$$

  By definition of $\lceil\cdot\rceil, \cdot \fatsemi \cdot$ we have:

  $$\lceil C_1 \fatsemi C_2\rceil = \lceil (I_1; C_1') \fatsemi C_2\rceil = \lceil I_1; (C_1' \fatsemi C_2)\rceil = \lceil I_1\rceil; \lceil C_1' \fatsemi C_2\rceil$$

Using lemma (B.1) we find:

$$\exists \Sigma^\heartsuit,\ \langle\lceil I_1\rceil, \Sigma, \mathscr{C}\rangle \twoheadrightarrow \langle \mathbf{0}, \Sigma^\heartsuit, \mathscr{C}\rangle \ \wedge\ \langle\lceil C_1' \fatsemi C_2\rceil, \Sigma^\heartsuit, \mathscr{C}\rangle \Downarrow \Sigma''$$

Thus we prove the goal by choosing $\Sigma' = \Sigma^\star$ and constructing the following execution (the extension of the semantics rule for $\cdot;\cdot$ to $\cdot \twoheadrightarrow \cdot$ is assumed as obvious):

$$\begin{array}{ll} & \langle\lceil I_1\rceil; \lceil C_1'\rceil, \Sigma, \mathscr{C}\rangle \twoheadrightarrow \langle \mathbf{0}; \lceil C_1'\rceil, \Sigma^\heartsuit, \mathscr{C}\rangle \\ & \quad \rightarrow \langle\lceil C_1'\rceil, \Sigma^\heartsuit, \mathscr{C}\rangle \twoheadrightarrow \langle \mathbf{0}, \Sigma^\star, \mathscr{C}\rangle \\ \wedge & \langle\lceil C_2\rceil, \Sigma^\star, \mathscr{C}\rangle \Downarrow \Sigma'' \end{array}$$

□

Appendix C

Helper Lemmas for Constraint Reconstruction

C.1 Monotonicity of $\phi_{\mathscr{C}}$

To define and prove monotonicity we first need to define an order relation on **ProcConstr**. We define it as follows:

$$\delta_1 \leq \delta_2 \quad \triangleq \quad \forall X,\ \delta_1\ X \subseteq \delta_2\ X$$

By unrolling the definition of $\phi_{\mathscr{C}}$ the reduce the proof of monotonicity to the following theorem:

$$\delta_1 \leq \delta_2\ \wedge\ \delta_1, \eta \vdash C \rhd E_1\ \wedge\ \delta_2, \eta \vdash C \rhd E_2 \quad \Longrightarrow \quad E_1 \subseteq E_2$$

Proof:
We proceed by structural induction on C. The majority of cases are trivial because the typing rules compute E_1, E_2 such that $E_1 = E_2$.
The only interesting case is $C = X(\vec{p})$. By inverting $\cdot \vdash \cdot \rhd \cdot$ we find:

$$E_{X,1} \subseteq E_{X,2}$$

Since $\subseteq$ is preserved by equal substitutions the conclusion holds. □

C.2 Rewriting η in $\underline{\text{cansolve}}$

We need to prove:

$$\begin{aligned} &\underline{\text{cansolve}}\ E[\eta/\eta' \sqcup \Psi]\ \Gamma\ pc \quad \wedge \quad (\forall pc', [\![\Psi]\!]\ \Gamma\ pc' = \ell) \\ &\Longrightarrow \underline{\text{cansolve}}\ E\ \Gamma\ (pc \sqcup \ell) \end{aligned}$$

Proof:
By unfolding the definition of cansolve we can reduce the theorem to the following:

$$[\![\Psi^\star[\eta/\eta' \sqcup \Psi]]\!]\ \Gamma\ pc \sqsubseteq \Gamma\ q.y\ \wedge (\forall pc', [\![\Psi]\!]\ \Gamma\ pc' = \ell) \Longrightarrow\ [\![\Psi^\star]\!]\ \Gamma\ (pc \sqcup \ell) \sqsubseteq \Gamma\ q.y$$

We proceed by induction on $\Psi^\star$:

- *case bottom:* Trivial
- *case p.x:*
$$[\![p.x[\eta/\cdot]]\!]\ \Gamma\ pc = [\![p.x]\!]\ \Gamma\ pc' = \Gamma\ p.x$$
- *case η:*
$$[\![\eta[\eta/\eta' \sqcup \Psi]]\!]\ \Gamma\ pc = [\![\eta' \sqcup \Psi]\!]\ \Gamma\ pc = [\![\eta']\!]\ \Gamma\ pc\ \sqcup\ [\![\Psi]\!]\ \Gamma\ pc = pc \sqcup \ell$$
While simultaneously:
$$[\![\eta]\!]\ \Gamma\ (pc \sqcup \ell) = pc \sqcup \ell$$
- *case sup:* This case follows easily using the inductive hypothesis,
$[\![\Psi_1^\star \sqcup \Psi_2^\star]\!] = [\![\Psi_1^\star]\!] \sqcup [\![\Psi_2^\star]\!]$ and $(\Psi_1^\star \sqcup \Psi_2^\star)[\eta/\cdot] = \Psi_1^\star[\eta/\cdot] \sqcup \Psi_2^\star[\eta/\cdot]$

C.3 Substitution in cansolve

We need to show:

$$\underline{\text{cansolve}}\ E[\vec{q}/\vec{p}\,]\ \Gamma\ pc \Longrightarrow \underline{\text{cansolve}}\ E\ \Gamma[\vec{q} \mapsto \vec{p}\,]\ pc$$

Proof:
For brevity we consider only the scalar case: the extension is trivial if $\vec{q}$ is taken sufficiently fresh (as assumed in 4.1). By unfolding cansolve we reduce the theorem to the following:

$$[\![\Psi[q/p]]\!]\ \Gamma\ pc \sqsubseteq \Gamma\ (r.y[q/p]) \Longrightarrow [\![\Psi]\!]\ \Gamma[q \mapsto p]\ pc \sqsubseteq \Gamma[q \mapsto p]\ r.y$$

We proceed by induction on the structure of Ψ:

- *case bottom:* Trivial
- *case s.x:* We have:
$$[\![s.x[q/p]]\!]\ \Gamma\ pc \sqsubseteq \Gamma\ (r.y[q/p])$$
Which by definition of $[\![\cdot]\!]$ can be rewritten into:
$$\Gamma\ (s.x[q/p]) \sqsubseteq \Gamma\ (r.y[q/p])$$

While simultaneously we have:

$$[\![s.x]\!]\ \Gamma[q \mapsto p]\ pc \sqsubseteq \Gamma[q \mapsto p]\ r.y$$

Which can be rewritten into:

$$\Gamma[q \mapsto p]\ s.x \sqsubseteq \Gamma[q \mapsto p]\ r.y$$

The equation of the two constraints follows from the following lemma:

$$\Gamma[q \mapsto p]\ r = \Gamma\ r[q/p]$$

Which was proven in *lean* as part of the proof of the unwinding lemma (4.14).

- *case η:* Trivial
- *case sup:* This case follows easily using the inductive hypothesis, $[\![\Psi_1^\star \sqcup \Psi_2^\star]\!] = [\![\Psi_1^\star]\!] \sqcup [\![\Psi_2^\star]\!]$ and $(\Psi_1^\star \sqcup \Psi_2^\star)[\eta/\cdot] = \Psi_1^\star[\eta/\cdot] \sqcup \Psi_2^\star[\eta/\cdot]$

Bibliography

[1] Ajlan Al-Ajlan. The comparison between forward and backward chaining. *International Journal of Machine Learning and Computing*, 5(2):106–113, 2015.

[2] 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.

[3] Jeremy Avigad, Leonardo De Moura, and Soonho Kong. Theorem proving in lean. 2021.

[4] Brian E Aydemir, Aaron Bohannon, Matthew Fairbairn, J Nathan Foster, Benjamin C Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In *International Conference on Theorem Proving in Higher Order Logics*, pages 50–65. Springer, 2005.

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

[6] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party session. *Logical Methods in Computer Science*, 8, 2012.

[7] Thierry Coquand and Gérard Huet. *The calculus of constructions*. PhD thesis, INRIA, 1986.

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

[9] Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. A formal theory of choreographic programming. *Journal of Automated Reasoning*, 67(2):21, 2023.

[10] Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

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

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

[13] John Harrison, Josef Urban, and Freek Wiedijk. History of interactive theorem proving. In *Handbook of the History of Logic*, volume 9, pages 135–214. Elsevier, 2014.

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

[15] William A Howard et al. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

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

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

[18] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. In *European Symposium on Research in Computer Security*, pages 97–110. Springer, 1998.

[19] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[20] The mathlib Community. mathlib4: The lean mathematical library for lean 4. GitHub repository, 2025. Accessed 2025-10-04.

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

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

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

[24] 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.

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

[26] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

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

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

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

[30] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. 1955.

[31] 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.

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

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

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

[35] 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.

Acknowledgments

Thank you all