UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

Dipartimento di Matematica

Formalizing the Operational Semantics of the π-Calculus

A Solution to the Concurrent Calculi Formalization Benchmark (part 2)

Relatore: prof. Alberto Momigliano

Tesi di Laurea di: Gabriele Cecilia

Anno accademico 2022/2023

Abstract

This thesis presents a formalization of two different operational semantics of the π-calculus within the proof environment Beluga and provides a solution to the second challenge of the Concurrent Calculi Formalization Benchmark. Specifically, this challenge involves estabilishing a correspondence between silent transitions in the labelled transition system semantics and reductions in the reduction semantics of the π-calculus. The proof environment employed, Beluga, has been developed at McGill University in Montreal, Canada, and allows specification of formal systems using a foundation of contextual modal logic.

To begin, we offer an overview of the role of formalization in program verification and in the metatheory of programming languages. We also delve into the details of the Concurrent Benchmark, focusing on its purpose of exploring the state of the art in the mechanization of process calculi, finding the best practices to address their typical issues and improving the tools for their encoding.

Next we introduce the syntax and semantics for the subset of the π-calculus featured in the challenge, following the existing literature (particularly, "The π-calculus - a Theory of Mobile Processes" by Sangiorgi & Walker). Additionally, we provide a comprehensive informal proof of the theorems regarding semantics equivalence.

Subsequently, we present a HOAS formalization in Beluga of the aforementioned results. Definitions and theorems of the benchmark problem are introduced together with a gradual explanation of the Beluga constructs and features used. Furthermore, we address the challenges encountered during the encoding process and the corresponding solutions devised.

Finally, we draw the conclusions of the accomplished work. We provide an evaluation of the choice of the proof assistant Beluga, together with a discussion about the contributions of our work: in summary, this thesis offers a first formalization of semantics equivalence in the π-calculus and introduces a couple of useful encoding techniques for specific constructs, such as telescopes and induction over two arguments. Additionally, we present a brief overview of some related pre-existing works and some potential directions for future research studies.

Contents

1 Introduction **4**

2 The π-Calculus and its Operational Semantics **7**
- 2.1 Syntax . . . 7
  - 2.1.1 Bound Names and Variable Convention . . . 8
- 2.2 Semantics . . . 9
  - 2.2.1 Reduction Semantics . . . 9
  - 2.2.2 Labelled Transition System Semantics . . . 10
- 2.3 Equivalence of Reduction and LTS Semantics . . . 13
  - 2.3.1 Theorem 1: τ-Transition Implies Reduction . . . 13
  - 2.3.2 Theorem 2: Reduction Implies τ-Transition . . . 18

3 Beluga Formalization **26**
- 3.1 Syntax . . . 26
- 3.2 Semantics . . . 28
  - 3.2.1 Reduction Semantics . . . 28
  - 3.2.2 Labelled Transition System Semantics . . . 29
- 3.3 Equivalence of Reduction and LTS Semantics . . . 31
  - 3.3.1 Theorem 1: τ-Transition Implies Reduction . . . 34
  - 3.3.2 Theorem 2: Reduction Implies τ-Transition . . . 42

4 Conclusions **53**
- 4.1 Evaluation . . . 53
- 4.2 Related and Future Work . . . 56

A Equivalence between Early and Late Semantics **60**

B Proof of Lemma 2.6 **66**

C Formalization of Lemma 2.6 **75**

Chapter 1

Introduction

Nowadays, programs and software are part of the daily life of every individual. The widespread use of smartphones, computers and smart technology in homes and workplaces is largely dependent on programming. The internet, social media platforms and satellite networks allow global communication, commerce and entertainment. Furthermore, emerging technologies such as artificial intelligence and machine learning are opening new development frontiers. Programming is transitioning from being a mastery performed only by experts into becoming a regular task involving a large community of developers.

As the production of new and increasingly advanced programs is becoming more and more significant, so does the need for verification of program correctness. Programming errors are fairly common and can occasionally lead to catastrophic events, resulting in loss of lives and financial damages. For instance, we mention the AT&T Network crash in 1990, where a bug in a C program caused the shutdown of the USA's largest telephone network, or the explosion of the Ariane 5 rocket in 1996, attributed to errors in the software design [17]. These incidents highlight the critical importance of implementing testing and verification procedures in software development.

While practices like delivering a thorough project documentation and dividing code development among different teams can contribute to better code quality, errors can still arise at any stage of software production. Formal methods offer a solution by enhancing software reliability throughout the entire development process. More specifically, the formalization of a program consists in providing a mathematical description of the program specifications and properties. This process may include modeling the system in a mathematical environment, defining program specifications, testing the software, and verifying its correctness through proofs. Formalization is typically executed through the use of a proof assistant, a software which provides the required mathematical infrastructure and usually has a graphical interface guiding the user towards the search for proofs with tactics or computation holes.

A niche domain of formal methods application is the metatheory of programming languages. Being able to reason about the language in which a software is written is necessary to verify static analysis tools (such as type systems), compilation and optimization. A well-known issue in this field lies in the representation of language constructs that bind variables: to address this problem, various techniques have been developed, such as De Bruijn indices [5], nominal techniques [7] and higher-order abstract syntax (HOAS) [18]. In order to raise general interest on the mechanization of programming languages and to devise better tools and strategies to address specific encoding issues, challenges like the POPLMark [1] or the Concurrent Calculi Formalization Benchmark [12] have been published throughout the years.

The POPLMark challenge [1] is a collection of benchmarks about programming language mechanization published in 2005. Its purpose was to measure the progress in formalizing programming language metatheory and to simplify proofs in this domain by finding the most appropriate techniques to solve recurring problems. The challenge successfully increased general interest in this field, with many researchers providing their own solutions to the benchmark problems.

Following in the footsteps of the POPLMark challenge, a new set of benchmarks related to concurrency mechanization [12] has been published. Concurrency is the branch of computer science in which multiple processes can be executed indipendently or interact with each other; process calculi are an example of mathematical models that allow the formalization of concurrent systems. The Concurrent Calculi Formalization Benchmark aims at exploring the state of the art in the formalization of process calculi, finding the best practices to address their typical issues and improving the tools for their mechanization.

The authors of the Concurrent Calculi Formalization Benchmark identified three key aspects which usually cause difficulties when mechanizing concurrency theory:

1. Linearity: the notion that a channel must be used exactly once in a process.

2. Scope extrusion: the fact that a process can send restricted names to another process, thus expanding their scope.

3. Coinductive reasoning: the mechanization of proofs regarding processes which have infinite behaviours, usually addressed with a technique called coinduction.

The authors designed three challenge problems which deal with each of these aspects separately. As a consequence, the comparison of different methods used to solve these problems is going to be more straightforward and effective.

This thesis provides a solution to the second problem of the benchmark, by formalizing the inherent definitions and demonstrations through the proof environment Beluga.

In order to formalize results regarding the metatheory of programming languages, the selected proof environment must provide a mathematical infrastructure capable of representing the language's syntax, defining the judgements describing its semantics, and allowing to formulate and prove theorems about it. Specifically, syntax is usually encoded through algebraic datatypes; semantics is encoded through inductive definitions and recursive functions; proofs are conducted using structural or well founded induction. However, a significant issue to be addressed consists in the method used to represent bindings of the object language [13].

One commonly used technique is based on *De Bruijn indices* [5]. In this approach, occurrences of a variable in a term are denoted by a natural number, representing the number of abstractions that must be traversed to bind that occurrence. De Bruijn indices offer the considerable advantage that each term has a unique representation, thus eliminating concerns about α-equivalence. However, interpreting de Bruijn indices may not be immediately intuitive, and the definition of capture-avoiding substitution requires careful handling of shifting indices.

Nominal techniques, introduced by Pitts and Gabbay [7], present an alternative approach. Given a countably infinite set of names $\mathcal{A}$, binders are represented by names that can be swapped via finite permutations over $\mathcal{A}$. α-equivalence is defined in terms of invariance of names under specific permutations, and capture-avoiding substitution is replaced by permutations themselves. While nominal techniques do not present a unique representation of terms, the large use of finite permutations simplifies mechanization aspects due to their favorable algebraic properties.

In *higher-order abstract syntax* [18], binders of the object language are represented by binders occurring in functions of the metalanguage. The major advantage of this approach is that issues related to α-renaming and substitution are delegated to the metalanguage developers: the formalizer inherits the solutions of these problems directly from the metalanguage [20], thereby avoiding the need to state and prove technical lemmas about binders, as required by other approaches. This is the main reason which led us to the choice of the HOAS technique for the formalization of π-calculus in this thesis.
While general-purpose proof envoronments such as Coq are built on powerful dependent type theories and can implement HOAS, they lack direct support for many intricate aspects regarding mechanization of formal systems, often requiring users to rely on techniques or libraries to address them [20]. Therefore, we opted for the proof assistant Beluga ([19], [20], [21], [22]), which sacrifices some of the computational power present in Coq but provides a sophisticated infrastructure for handling variables and assumptions automatically, thus streamlining the mechanization process. In Chapter 3 we will provide an overview of the features of this proof assistant, along with the formalization of our work.

Chapter 2

The π-Calculus and its Operational Semantics

The π-calculus is an example of process calculus, a formal mathematical framework for modeling and reasoning about concurrent systems. Processes are the fundamental units of computation and they interact with each other through channels. An essential feature of the π-calculus is that processes can transfer channel names to each other, transforming their interaction network; this renders the π-calculus suitable to describe systems whose interconnections vary over time.

More specifically, the π-calculus allows declaring that a certain channel name is local to some process; this means that it can be used by this process only. However, the channel name can be sent to some other process: the result is that it becomes local to both processes, and each of them can use it. The phenomenon of the expansion of the scope of a channel name is referred to as "scope extrusion".

Formalizing semantically correct operational rules that encode scope extrusion is typically a challenging issue. Historically, researchers have developed various mechanizations of scope extrusion, each managing α-conversion or binders differently. Additionally, alternative approaches which avoid dealing with scope extrusion have been elaborated.

The second challenge of the Concurrent Calculi Formalization Benchmark [12] involves reasoning about this topic. The language used in this challenge is a simplified version of the π-calculus without constructs for match, replication and sums. The challenge requires defining two distinct operational semantics for this calculus - a labelled transition system (LTS) semantics, which directly addresses scope extrusion, and a reduction semantics, which circumvents it - and proving that they are equivalent up to structural congruence. This chapter presents the description of the language and the statement and proof of the theorems regarding semantics equivalence.

2.1 Syntax

We assume the existence of a countably infinite set of *names*, represented by the symbols $x, y, \ldots$. The set of *processes* Proc is defined by the following syntax:

$$P, Q \ := \ \mathbf{0} \ \mid \ x(y).P \ \mid \ \bar{x}y.P \ \mid \ P \mid Q \ \mid \ (\nu x)P$$

The symbol $\mathbf{0}$ refers to the empty process, which cannot perform any transition. The *input prefix* $x(y).P$ represents a process which can receive a name through the channel x and then behave as P, with the newly received name replacing y in P. The *output prefix* $\bar{x}y.P$ corresponds to a process which can send the name y through the channel x and then behave as P. The *parallel composition* $P \mid Q$ represents a process where both components P and Q can act indipendently or interact with each other. The *restriction* $(\nu x)P$ acts as the process P, but the name x is declared to be local to the process P and its usage is restricted to P only.

The input prefix $x(y).P$ and the restriction $(\nu y)P$ both bind the name y in P; in such cases, we can say that any occurrence of y in P is bound. Any other occurrence of names in processes is said to be free. More specifically, we can define the set of free names occurring in a process P, denoted as $\mathsf{fn}(P)$, and the set of bound names occurring in a process P, denoted as $\mathsf{bn}(P)$, by induction on the structure of processes as follows:

$$\begin{aligned}
\mathsf{fn}(\mathbf{0}) &= \emptyset & \mathsf{bn}(\mathbf{0}) &= \emptyset\\
\mathsf{fn}(x(y).P) &= \{x\} \cup (\mathsf{fn}(P) \setminus \{y\}) & \mathsf{bn}(x(y).P) &= (\{y\} \cap \mathsf{fn}(P)) \cup \mathsf{bn}(P)\\
\mathsf{fn}(\bar{x}y.P) &= \{x, y\} \cup \mathsf{fn}(P) & \mathsf{bn}(\bar{x}y.P) &= \mathsf{bn}(P)\\
\mathsf{fn}(P \mid Q) &= \mathsf{fn}(P) \cup \mathsf{fn}(Q) & \mathsf{bn}(P \mid Q) &= \mathsf{bn}(P) \cup \mathsf{bn}(Q)\\
\mathsf{fn}((\nu x)P) &= \mathsf{fn}(P) \setminus \{x\} & \mathsf{bn}((\nu x)P) &= (\{x\} \cap \mathsf{fn}(P)) \cup \mathsf{bn}(P)
\end{aligned}$$

Figure 2.1: Definition of free and bound names in processes.

2.1.1 Bound Names and Variable Convention

In languages featuring bound variables, terms which differ only by the names assigned to bound variables are called *α-convertible* terms [4]. Such terms carry the same meaning and for this reason they are often identified. In the informal description of a language, α-conversion is sometimes introduced in a mathematically rigorous way, for example by partitioning the set of terms and considering its equivalence classes. In some cases, it is introduced as part of the language's semantics. In other situations, developers include it in some variable conventions, such as the Barendregt conventions for the λ-calculus [2]. Variable conventions might be difficult to justify formally (see [25] for an example) but often lead to simpler proofs and less reasoning about variables. Such different informal approaches to the representation of bound variables result in different formalizations as well.

This holds particularly true for the π-calculus, where a variety of historical approaches to this issue have led to different presentations of the calculus. For example, Parrow [16] explicitly incorporates α-equivalence into the semantics as a congruence rule. Milner et al. [15] or Honsell et al. [10] do not directly identify α-convertible processes; instead, they integrate process identification within LTS semantics rules. Sangiorgi & Walker [23] adopt a different approach, relying on two variable conventions: one stating that "processes that are α-convertible are identified", and another stating that "we assume that the bound names of any processes or actions under consideration are chosen to be different from the names free in any other entities under consideration". These conventions entail taking the quotient of the set of processes to include only one instance of α-convertible processes and disallowing processes with both free and bound occurrences of the same name.

Our approach for this thesis consists in adopting a less restrictive version of the variable conventions mentioned above. Specifically we assert that, given a process, *it is possible* to α-rename the bound occurrences of variables within it; additionally, we state that the bound names of any processes or actions under consideration *can* be chosen different from the names free in any other entities under consideration. Unlike Sangiorgi & Walker, we do not exclude processes with both free and bound occurrences of the same name; instead, we permit to α-convert such processes at need. While we are not going to build a mathematical framework to justify these conventions, we believe they offer an intuitive vision of the role of binders in processes and observe that they will play a significant role in the theorem proofs.

2.2 Semantics

2.2.1 Reduction Semantics

In reduction semantics, structural congruence relates "syntactically" equivalent processes; then, processes are reduced up to syntactic equivalence. Reduction does not explicitly handle scope extrusion; instead, it implicitly relies on congruence to address this aspect.

Structural Congruence

We define the *congruence* relation $\equiv$ as the smallest relation over Proc, closed under compatibility ($\star$) and equivalence laws ($\star\star$), satisfying the following axioms:

$$\overset{\textsc{Par-Assoc}}{\dfrac{}{P \mid (Q \mid R) \;\equiv\; (P \mid Q) \mid R}} \qquad \overset{\textsc{Par-Unit}}{\dfrac{}{P \mid \mathbf{0} \;\equiv\; P}} \qquad \overset{\textsc{Par-Comm}}{\dfrac{}{P \mid Q \;\equiv\; Q \mid P}}$$

$$\overset{\textsc{Sc-Ext-Zero}}{\dfrac{}{(\nu x)\,\mathbf{0} \;\equiv\; \mathbf{0}}} \qquad \overset{\textsc{Sc-Ext-Par}}{\dfrac{x \notin \mathsf{fn}(Q)}{(\nu x)P \mid Q \;\equiv\; (\nu x)(P \mid Q)}} \qquad \overset{\textsc{Sc-Ext-Res}}{\dfrac{}{(\nu x)(\nu y)P \;\equiv\; (\nu y)(\nu x)P}}$$

$(\star)$

$$\overset{\textsc{C-In}}{\dfrac{P \equiv Q}{x(y).P \;\equiv\; x(y).Q}} \qquad \overset{\textsc{C-Out}}{\dfrac{P \equiv Q}{\bar{x}y.P \;\equiv\; \bar{x}y.Q}} \qquad \overset{\textsc{C-Par}}{\dfrac{P \equiv P'}{P \mid Q \;\equiv\; P' \mid Q}} \qquad \overset{\textsc{C-Res}}{\dfrac{P \equiv Q}{(\nu x)P \;\equiv\; (\nu x)Q}}$$

$(\star\star)$

$$\overset{\textsc{C-Ref}}{\dfrac{}{P \equiv P}} \qquad \overset{\textsc{C-Sym}}{\dfrac{P \equiv Q}{Q \equiv P}} \qquad \overset{\textsc{C-Trans}}{\dfrac{P \equiv Q \qquad Q \equiv R}{P \equiv R}}$$

Figure 2.2: Structural congruence rules.

The first three rules are commutative monoid axioms for parallel composition. The second set of three rules involves scope extrusion and essentially asserts that the specific placement of binders is relatively unimportant, provided they refer to the same occurrences: for instance, the Sc-Ext-Par rule states that the processes $(\nu x)P \mid Q$ (where the scope of the name x is restricted to P) and $(\nu x)(P \mid Q)$ (where the scope of x is extended to $P \mid Q$) are syntactically equivalent, if x does not occur freely in Q. The compatibility rules ($\star$) ensure that processes with congruent subprocesses are also congruent, while equivalence rules ($\star\star$) establish congruence as an equivalence relation.

Note that a second C-Par rule with a different premise (congruence of processes on the right side of parallel compositions) is not needed. This is because it can be derived through a combination of monoid axioms and equivalence rules.

Reduction

We define the *reduction* relation $\to$ as the smallest relation over Proc satisfying the following rules:

$$\textsc{R-Com}\quad \frac{}{\bar{x}y.P \mid x(z).Q \;\to\; P \mid Q\{y/z\}} \qquad\qquad \textsc{R-Par}\quad \frac{P \to Q}{P \mid R \;\to\; Q \mid R}$$

$$\textsc{R-Res}\quad \frac{P \to Q}{(\nu x)P \;\to\; (\nu x)Q} \qquad\qquad \textsc{R-Struct}\quad \frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}$$

Figure 2.3: Reduction rules.

The R-Com rule performs the interaction of a pair of input and output prefix processes exchanging some name through the same channel; the notation $Q\{y/z\}$ represents capture-avoiding substitution of y for z in the process Q. The R-Par and R-Res rules ensure that reduction can proceed underneath a parallel composition or restriction. The R-Struct rule states that congruent processes can be reduced to congruent processes.

Note that a second R-Par rule with a different premise (reduction of processes on the right side of parallel compositions) is not needed. This is because it can be derived through a R-Struct rule.

2.2.2 Labelled Transition System Semantics

In LTS semantics, process behaviour is given by transitions: processes make transitions through actions, returning another process. Transitions represent changes in the state of processes following input/output or interaction with other processes. We present the early LTS semantics introduced in the challenge.

Early Semantics

The set of *actions* Act is defined by the following syntax:

$$\alpha := x(y) \mid \bar{x}y \mid \bar{x}(y) \mid \tau$$

The *input action* $x(y)$ represents the input of the name y through the channel x. The *free output action action* $\bar{x}y$ represents the output of the free name y through the channel x. The *bound output action* $\bar{x}(y)$ represents the output of the bound name y through the channel x. The *internal action* τ represents communication between two processes exchanging names.

Analogously to processes, we can define bound and free occurrences of names in an action. Given an input action $x(y)$ or a free output action $\bar{x}y$, we can say that both x and y occur free; given a bound output action $\bar{x}(y)$, x occurs free and y occurs bound; there are no name occurrencies in an internal action τ. We denote as $\mathsf{bn}(\alpha)$, $\mathsf{fn}(\alpha)$ and $\mathsf{n}(\alpha)$ the sets of *bound names*, *free names* and *names* occurring in an action α respectively.

We define the *transition* relation $\xrightarrow{(-)}$ as the smallest subset of $\mathsf{Proc} \times \mathsf{Act} \times \mathsf{Proc}$ which satisfies the following rules:

$$\textsc{S-In}\quad \frac{}{x(z).P \xrightarrow{x(y)} P\{y/z\}} \qquad\qquad \textsc{S-Out}\quad \frac{}{\bar{x}y.P \xrightarrow{\bar{x}y} P}$$

$$\textsc{S-Par-L}\quad \frac{P \xrightarrow{\alpha} P' \qquad \mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad\qquad \textsc{S-Par-R}\quad \frac{Q \xrightarrow{\alpha} Q' \qquad \mathsf{bn}(\alpha) \cap \mathsf{fn}(P) = \emptyset}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\textsc{S-Com-L}\quad \frac{P \xrightarrow{\bar{x}y} P' \qquad Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad\qquad \textsc{S-Com-R}\quad \frac{P \xrightarrow{x(y)} P' \qquad Q \xrightarrow{\bar{x}y} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\textsc{S-Res}\quad \frac{P \xrightarrow{\alpha} P' \qquad z \notin \mathsf{n}(\alpha)}{(\nu z)P \xrightarrow{\alpha} (\nu z)P'} \qquad\qquad \textsc{S-Open}\quad \frac{P \xrightarrow{\bar{x}z} P' \qquad z \neq x}{(\nu z)P \xrightarrow{\bar{x}(z)} P'}$$

$$\textsc{S-Close-L}\quad \frac{P \xrightarrow{\bar{x}(z)} P' \quad Q \xrightarrow{x(z)} Q' \quad z \notin \mathsf{fn}(Q)}{P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')} \qquad\qquad \textsc{S-Close-R}\quad \frac{P \xrightarrow{x(z)} P' \quad Q \xrightarrow{\bar{x}(z)} Q' \quad z \notin \mathsf{fn}(P)}{P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')}$$

Figure 2.4: Early transition semantics rules.

The S-In and S-Out rules describe transitions of input/output prefix processes. The two S-Par rules represent the independent progression of one of the two processes in a parallel composition. We note that the side condition $\mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset$, which ensures that no undesired variable captures occur, is necessary in this presentation; however, it could be omitted if more restrictive variable conventions were employed. The two S-Com rules represent interaction of the two processes in a parallel composition; since we are describing the early semantics, no substitution is present in the conclusion, as it is assumed to have been performed earlier through a S-In rule. The S-Res rule represents transition of a restriction through some action α, provided the restricted name does not appear in α. The S-Open rule introduces bound output transitions. The two S-Close rules describe the scenario in which a channel name, initially restricted to one component of a parallel composition, is sent to the other component: the result is a process where the scope of the restricted name is extended to the entire parallel composition. We observe that, in the bound output transition $P \xrightarrow{\bar{x}(z)} P'$, the name z is bound in the action $\bar{x}(z)$ and in the process P but may occur free in the process P': this fact is necessary to be able to express scope extrusion. Although it does not contradict our variable convention, we note that a more restrictive convention would require an exception to allow both free and bound occurrences of the same name.

Late Semantics

While the early semantics has been presented in the previous section, it is worth noting that there are other versions of labelled transition system semantics definitions for the π-calculus in the literature. One common alternative is the *late* semantics, which presents a different approach to modeling the behavior of processes. In contrast to the early semantics, the late semantics delays the substitution of names received in interactions as much as possible, namely during the execution of the S-Com rule. "It is a matter of taste which semantics to adopt" [16], as they are equivalent (see Appendix A for a formal proof of the equivalence). In the present paragraph we are going to outline the corresponding modifications of the late semantics with respect to the early semantics.

The syntax of the set of actions Act is the same as in the early semantics. There is a slight change in the definition of the sets $\mathsf{bn}(\alpha)$ and $\mathsf{fn}(\alpha)$: unlike the previous case, the name y occurring in an input action $x(y)$ is considered bound.

Regarding transitions, the rules S-In, the two S-Com rules and the two S-Close are replaced by the following:

S-In

$$\frac{}{x(z).P \xrightarrow{x(z)} P}$$

S-Com-L

$$\frac{P \xrightarrow{\bar{x}y} P' \qquad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{y/z\}}$$

S-Com-R

$$\frac{P \xrightarrow{x(z)} P' \qquad Q \xrightarrow{\bar{x}y} Q'}{P \mid Q \xrightarrow{\tau} P'\{y/z\} \mid Q'}$$

S-Close-L

$$\frac{P \xrightarrow{\bar{x}(z)} P' \qquad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')}$$

S-Close-R

$$\frac{P \xrightarrow{x(z)} P' \qquad Q \xrightarrow{\bar{x}(z)} Q'}{P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')}$$

Figure 2.5: Late transition semantics rules.

In the S-IN rule, there is no substitution of the input name within the process P: instead, the variable z, which might occur free in P, serves as a placeholder for the name that will later fill such spot. Only when an input prefix and output prefix communicate through a S-COM rule, the actual input name is deduced and then substituted. Regarding the S-CLOSE rules, the early semantics included a side condition $z \notin \mathsf{fn}(Q)$, with Q being the process performing the input transition: in case the name z received as input already occurred free in Q, then all the occurrences of z in the resulting process $(\nu z)(P' \mid Q')$ would be bound, thus leading to undesired variable capture. However, since the late version of the S-IN rule does not perform any substitution, there is no input name to be careful about and the side condition in the late S-CLOSE rules is not needed anymore.

2.3 Equivalence of Reduction and LTS Semantics

The goal of the challenge consists in proving the following theorems:

Theorem 1. *$P \xrightarrow{\tau} Q$ implies $P \to Q$.*

Theorem 2. *$P \to Q$ implies the existence of a Q' such that $P \xrightarrow{\tau} Q'$ and $Q \equiv Q'$.*

The first theorem states that every transition through a τ action corresponds to a reduction. The second theorem states that, given a reduction of the process P to the process Q, P is able to make a τ-transition to some process Q' congruent to Q. Thus, these theorems show that these two semantics are equivalent up to structural congruence. In this section we outline the key steps of the proof for the two theorems, providing the full details of one extensive proof in Appendix B.

2.3.1 Theorem 1: τ-Transition Implies Reduction

Before proving the first theorem, we state three preliminary lemmas which describe rewriting (up to structural congruence) of processes involved in specific transitions.

Lemma 1.1. ***Rewriting of processes involved in input transitions***
If $Q \xrightarrow{x(y)} Q'$ then there exist a finite (possibly empty) set of names $w_1, ..., w_n$ (with $x, y \neq w_i\ \forall i = 1, ..., n$) and two processes R, S such that
$Q \equiv (\nu w_1)...(\nu w_n)(x(z).R \mid S)$ and $Q' \equiv (\nu w_1)...(\nu w_n)(R\{y/z\} \mid S)$.

Lemma 1.2. ***Rewriting of processes involved in free output transitions***
If $Q \xrightarrow{\bar{x}y} Q'$, then there exist a finite (possibly empty) set of names $w_1, ..., w_n$ (with $x, y \neq w_i\ \forall i = 1, ..., n$) and two processes R, S such that
$Q \equiv (\nu w_1)...(\nu w_n)(\bar{x}y.R \mid S)$ and $Q' \equiv (\nu w_1)...(\nu w_n)(R \mid S)$.

Lemma 1.3. ***Rewriting of processes involved in bound output transitions***
If $Q \xrightarrow{\bar{x}(z)} Q'$, then there exist a finite (possibly empty) set of names $w_1, ..., w_n$ (with $x \notin \{z, w_1, ..., w_n\}$) and two processes R, S such that
$Q \equiv (\nu z)(\nu w_1)...(\nu w_n)(\bar{x}z.R \mid S)$ and $Q' \equiv (\nu w_1)...(\nu w_n)(R \mid S)$.

We prove Lemma 1.1.

Proof. Let $Q \xrightarrow{x(y)} Q'$ (T1). We proceed by induction on the structure of Q.

- $Q = \mathbf{0}$: by hyphotesis $\mathbf{0} \xrightarrow{x(y)} Q'$, but the process $\mathbf{0}$ does not undergo any transition, so we have a contradiction.

- $Q = w(z).P$: by inversion on the transition (T1) through the rule S-In, we obtain that $w = x$ and $Q' = P\{y/z\}$; the transition (T1) can be rewritten as $x(z).P \xrightarrow{x(y)} P\{y/z\}$.

  We observe that $x(z).P \equiv x(z).P \mid \mathbf{0}$ by Par-Unit and $P\{y/z\} \equiv P\{y/z\} \mid \mathbf{0}$ also by Par-Unit. Therefore, taking an empty set of names and setting $R := P$ and $S := 0$, the conclusion holds.

- $Q = \bar{w}z.P$: by hypothesis $\bar{w}z.P \xrightarrow{x(y)} Q'$, but this process can only undergo transitions via free output actions, so we have a contradiction.

- $Q = Q_1 \mid Q_2$: by inversion on $Q_1 \mid Q_2 \xrightarrow{x(y)} Q'$ we obtain two subcases depending on the transition rule used.

  – S-Par-L: the transition (T1) can be rewritten as $Q_1 \mid Q_2 \xrightarrow{x(y)} R_1 \mid Q_2$; we also have that $y \notin \text{fn}(Q_2)$ and that $Q_1 \xrightarrow{x(y)} R_1$ (T2).

  Applying the inductive hypothesis to (T2), we obtain that there exist names $w_1, ..., w_n$ different from x, y and processes R, S such that $Q_1 \equiv (\overline{\nu w})(x(z).R \mid S)$, denoted as (H1), and $R_1 \equiv (\overline{\nu w})(R\{y/z\} \mid S)$, denoted as (H2), with $(\overline{\nu w})$:= $(\nu w_1)...(\nu w_n)$. For the variable convention, we can assume that the w_i are not free in Q_2. Finally, through a chain of congruences we can show that $Q_1 \mid Q_2$ and $R_1 \mid Q_2$ have the desired form, thus proving the thesis.

  We consider $Q_1 \mid Q_2$. First we take (H1) and apply the C-Par rule:

$$\text{C-Par}\ \frac{Q_1 \ \equiv\ (\overline{\nu w})(x(z).R \mid S) \quad \text{(H1)}}{Q_1 \mid Q_2 \ \equiv\ (\,(\overline{\nu w})(x(z).R \mid S)\,) \mid Q_2 \quad \text{(C1)}}$$

  We then work on the right hand side of the congruence (C1): since the names w_i do not occur free in Q_2, we can apply the Sc-Ext-Par rule n times, in combination with the C-Trans rule. At the end of this process, we obtain the following congruence, denoted as (C2):
  $(\,(\overline{\nu w})(x(z).R \mid S)\,) \mid Q_2 \ \equiv\ (\overline{\nu w})(\,(x(z).R \mid S) \mid Q_2\,)$.
  By transitivity - i.e. by applying C-Trans to congruences (C1) and (C2) - we obtain that $Q_1 \mid Q_2 \ \equiv\ (\overline{\nu w})(\,(x(z).R \mid S) \mid Q_2\,)$ (C3).

  We finally work on the right hand side of the congruence (C3). First we consider the process $(\,(x(z).R \mid S) \mid Q_2\,)$, which is embedded in n restrictions, and apply the C-Sym and Par-Assoc rules:

$$\text{C-Sym}\ \frac{\text{Par-Assoc}\ \dfrac{}{x(z).R \mid (S \mid Q_2) \ \equiv\ (x(z).R \mid S) \mid Q_2}}{(x(z).R \mid S) \mid Q_2 \ \equiv\ x(z).R \mid (S \mid Q_2)}$$

  By applying the C-Res rule n times, we obtain the congruence $(C4)$:

$(\overline{\nu w})(\,(x(z).R \mid S) \mid Q_2\,) \;\equiv\; (\overline{\nu w})(\,x(z).R \mid (S \mid Q_2)\,)$.
By transitivity on the congruences (C3) and (C4) we obtain the desired congruence:

$$\textsc{C-Trans}\ \frac{(\mathrm{C3}) \qquad (\mathrm{C4})}{Q_1 \mid Q_2 \;\equiv\; (\overline{\nu w})(\,x(z).R \mid (S \mid Q_2)\,)}$$

The procedure for $R_1 \mid Q_2$ is analogous: starting from (H2) instead of (H1), we obtain that $R_1 \mid Q_2 \equiv (\overline{\nu w})(\,R\{y/z\} \mid (S \mid Q_2)\,)$.

– S-Par-R: the proof is similar to the previous case, requiring some additional application of monoid axioms for parallel composition.

- $Q = (\nu z)P$: by inversion on the transition $(\nu z)P \xrightarrow{x(y)} Q'$ through the rule S-Res, we obtain that $Q' = (\nu z)P'$, $z \notin \{x, y\}$ and that $P \xrightarrow{x(y)} P'$ (T2); the transition (T1) can be rewritten as $(\nu z)P \xrightarrow{x(y)} (\nu z)P'$.

  Applying the inductive hypothesis to the transition (T2), we obtain that there exist names $w_1, ..., w_n$ different from x, y (and z, up to renaming) and processes R, S such that $P \equiv (\overline{\nu w})(\,x(u).R \mid S\,)$ and $P' \equiv (\overline{\nu w})(\,R\{y/u\} \mid S\,)$.
  Through a single instance of the C-Res rule, we deduce that $(\nu z)P \equiv (\nu z)(\overline{\nu w})(\,x(u).R \mid S\,)$ and $(\nu z)P' \equiv (\nu z)(\overline{\nu w})(\,R\{y/u\} \mid S\,)$, hence the conclusion.

□

Lemmas 1.2 and 1.3 are proved by induction on the structure of processes as well, with a different base case. The base case for Lemma 1.2 consists in the output prefix case, analyzed through the S-Out rule. The base case for Lemma 1.3 consists in restriction, analyzed through the S-Open rule; we observe that the presence of a free output transition as an hypothesis requires the application of Lemma 1.2. Since the proof structure is essentially the same, details are omitted.

Finally, we prove Theorem 1. In order to improve readability, we are going to cover the most significant steps, leaving details to the reader for cases which do not offer anything new. We recall the statement of the theorem:

Theorem 1. *$P \xrightarrow{\tau} Q$ implies $P \rightarrow Q$.*

Proof. Let $P \xrightarrow{\tau} Q$ (T1). We proceed by induction on the structure of this transition.

- S-Par-L: the transition (T1) is rewritten as $P_1 \mid R \xrightarrow{\tau} Q_1 \mid R$; we have that $P_1 \xrightarrow{\tau} Q_1$ (T2).

  We apply the inductive hypothesis to the transition (T2), obtaining $P_1 \rightarrow Q_1$. Through the rule R-Par, we finally obtain $P_1 \mid R \rightarrow Q_1 \mid R$.

- S-Par-R: the transition (T1) is rewritten as $R \mid P_2 \xrightarrow{\tau} R \mid Q_2$; we have that $P_2 \xrightarrow{\tau} Q_2$ (T2).

As in the previous case, we apply the inductive hypothesis to the transition (T2), obtaining $P_2 \to Q_2$. Through the rule R-Par, we obtain that $P_2 \mid R \to Q_2 \mid R$. Finally we conclude by applying a R-Struct:

$$\frac{R \mid P_2 \equiv P_2 \mid R \qquad P_2 \mid R \to Q_2 \mid R \qquad Q_2 \mid R \equiv R \mid Q_2}{R \mid P_2 \to R \mid Q_2}$$

- S-Com-L: the transition (T1) is rewritten as $P_1 \mid P_2 \xrightarrow{\tau} Q_1 \mid Q_2$; we have that $P_1 \xrightarrow{\bar{x}y} Q_1$ and that $P_2 \xrightarrow{x(y)} Q_2$.

  We apply Lemmas 1.1 and 1.2 to rewrite the processes involved in these input and output transitions, obtaining the following congruences:

$$P_1 \equiv (\overline{\nu w})(\,\bar{x}y.R_1 \mid S_1\,); \qquad P_2 \equiv (\overline{\nu v})(\,x(z).R_2 \mid S_2\,);$$
$$Q_1 \equiv (\overline{\nu w})(\,R_1 \mid S_1\,); \qquad Q_2 \equiv (\overline{\nu v})(\,R_2\{y/z\} \mid S_2\,).$$

  Like before, $(\overline{\nu w})$ denotes $(\nu w_1)...(\nu w_n)$ and $(\overline{\nu v})$ denotes $(\nu v_1)...(\nu v_m)$. We can assume, up to α-renaming, that each w_i is different from v_j, that each w_i does not occur freely in the process $(\,x(z).R_2 \mid S_2\,)$ (as a consequence, w_i does not occur freely in $R_2\{y/z\} \mid S_2$ as well, since $w_i \neq y$ for Lemma 1.1), and that each v_j does not occur freely in $(\,\bar{x}y.R_1 \mid S_1\,)$ (hence in $R_1 \mid S_1$).
  Through the rule R-Com, we obtain the following reduction (R1):

$$\text{R-Com}\ \frac{}{\bar{x}y.R_1 \mid x(z).R_2 \to R_1 \mid R_2\{y/z\}}$$

  We then work on the processes involved in this congruence, gradually adding the subprocesses and binders present in P_i and Q_i, with the goal of reaching a reduction between processes congruent to $P_1 \mid P_2$ and $Q_1 \mid Q_2$ and then applying a R-Struct rule.

  First of all, we apply the R-Par rule to the reduction (R1) to add the process S_1; we obtain that $(\,\bar{x}y.R_1 \mid x(z).R_2\,) \mid S_1 \to (\,R_1 \mid R_2\{y/z\}\,) \mid S_1$.
  By using the monoid axioms for parallel composition, it can be shown that $(\,\bar{x}y.R_1 \mid x(z).R_2\,) \mid S_1$ is congruent to $(\,\bar{x}y.R_1 \mid S_1\,) \mid x(z).R_2$, and that $(\,R_1 \mid R_2\{y/z\}\,) \mid S_1$ is congruent to $(\,R_1 \mid S_1\,) \mid R_2\{y/z\}$. Applying the R-Struct rule, we obtain that $(\,\bar{x}y.R_1 \mid S_1\,) \mid x(z).R_2 \to (\,R_1 \mid S_1\,) \mid R_2\{y/z\}$.

  Similarly, we add S_2 to the right and obtain the following reduction:
  $(\,\bar{x}y.R_1 \mid S_1\,) \mid (\,x(z).R_2 \mid S_2\,) \to (\,R_1 \mid S_1\,) \mid (\,R_2\{y/z\} \mid S_2\,)$.

  Then, we apply the R-Res rule to introduce the restriction (νw_n) on both terms, obtaining the following reduction:
  $(\nu w_n)(\,(\bar{x}y.R_1 \mid S_1) \mid (x(z).R_2 \mid S_2)\,) \to (\nu w_n)(\,(R_1 \mid S_1) \mid (R_2\{y/z\} \mid S_2)\,)$.
  Now, since w_n does not occur freely in $(x(z).R_2 \mid S_2)$ or $(R_2\{y/z\} \mid S_2)$, we reduce the scope of w_n and move (νw_n) to the left process of the parallel compositions: we obtain that $(\nu w_n)(\,(\bar{x}y.R_1 \mid S_1) \mid (x(z).R_2 \mid S_2)\,)$ is congruent to $(\,(\nu w_n)(\bar{x}y.R_1 \mid S_1)\,) \mid (x(z).R_2 \mid S_2)$ through Sc-Ext-Par (same for the other process). Using R-Struct, we obtain that
  $(\,(\nu w_n)(\bar{x}y.R_1 \mid S_1)\,) \mid (x(z).R_2 \mid S_2) \to (\,(\nu w_n)(R_1 \mid S_1)\,) \mid (R_2\{y/z\} \mid S_2)$.

Similarly, we can introduce all the other binders. Thus, we obtain the reduction below, from which we reach the conclusion with a final R-Struct:

$$\begin{aligned}&(\,(\overline{\nu w})(\bar{x}y.R_1 \mid S_1)\,) \mid (\,(\overline{\nu v})(x(z).R_2 \mid S_2)\,) \rightarrow\\ &\rightarrow\ (\,(\overline{\nu w})\,(R_1 \mid S_1)\,) \mid (\,(\overline{\nu v})(R_2\{y/z\} \mid S_2)\,)\end{aligned}$$

- S-Com-R: analogous to the previous case, with some additional congruence and reduction steps due to the asymmetry of reduction rules.
- S-Res: the transition (T1) is rewritten as $(\nu z)P' \xrightarrow{\tau} (\nu z)Q'$; we have that $P' \xrightarrow{\tau} Q'$ (T2).

  We apply the inductive hypothesis to the transition (T2), obtaining $P' \rightarrow Q'$. Through the rule R-Res, we finally obtain $(\nu z)P' \rightarrow (\nu z)Q'$.
- S-Close-L: the transition (T1) is rewritten as $P_1 \mid P_2 \xrightarrow{\tau} (\nu z)(Q_1 \mid Q_2)$; we have that $P_1 \xrightarrow{\bar{x}(z)} Q_1$ and that $P_2 \xrightarrow{x(z)} Q_2$.

  We apply Lemmas 1.1 and 1.3 to rewrite the processes involved in these input and bound output transitions, obtaining the following congruences:

  $$\begin{aligned}&P_1 \equiv (\nu z)(\overline{\nu w})(\,\bar{x}z.R_1 \mid S_1\,); &&P_2 \equiv (\overline{\nu v})(\,x(y).R_2 \mid S_2\,);\\ &Q_1 \equiv (\overline{\nu w})(\,R_1 \mid S_1\,); &&Q_2 \equiv (\overline{\nu v})(\,R_2\{z/y\} \mid S_2\,).\end{aligned}$$

  As before, we can assume that each w_i is different from v_j, that each w_i does not occur freely in the two processes congruent to P_2 and Q_2, and that each v_j does not occur freely in the two processes congruent to P_1 and Q_1. Analogously to the case S-Com-L, we obtain the following reduction through R-Com:

  $$\text{R-Com}\ \frac{}{\bar{x}z.R_1 \mid x(y).R_2\ \rightarrow\ R_1 \mid R_2\{z/y\}}$$

  Then, we proceed exactly as in the case S-Com-L, first adding the processes S_1 and S_2, and then the restrictions, obtaining the following reduction:

  $$\begin{aligned}&(\,(\overline{\nu w})(\bar{x}z.R_1 \mid S_1)\,) \mid (\,(\overline{\nu v})(x(y).R_2 \mid S_2)\,) \rightarrow\\ &\rightarrow\ (\,(\overline{\nu w})\,(R_1 \mid S_1)\,) \mid (\,(\overline{\nu v})(R_2\{z/y\} \mid S_2)\,)\end{aligned}$$

  We exploit the fact that three of the processes present in the last reduction are congruent to P_2, Q_1, and Q_2 in order to obtain, through some congruence axioms and a R-Struct, the following reduction:

  $$(\,(\overline{\nu w})(\bar{x}z.R_1 \mid S_1)\,) \mid P_2\ \rightarrow\ Q_1 \mid Q_2$$

  We apply a rule R-Com to introduce (νz) in both sides of the reduction. Finally, since $z \notin \text{fn}(P_2)$, we can apply a Sc-Ext-Par to move (νz) to the left component of the parallel composition. We obtain the following reduction:

  $$(\,(\nu z)(\overline{\nu w})(\bar{x}z.R_1 \mid S_1)\,) \mid P_2\ \rightarrow\ (\nu z)(Q_1 \mid Q_2)$$

  Observing that the leftmost process is congruent to P_1 and applying an additional R-Struct, we come to the conclusion.
- S-Close-R: analogous to the previous case, with some additional congruence and reduction steps due to the asymmetry of reduction rules.

□

2.3.2 Theorem 2: Reduction Implies τ-Transition

Before proving the second theorem, we state and prove some intermediate results. We begin by reporting five minor lemmas regarding the existence of free and bound names in specific transitions.

Lemma 2.1. *If $P \xrightarrow{\bar{x}y} P'$, then $x, y \in$ fn(P).*

Proof. Let $P \xrightarrow{\bar{x}y} P'$; we proceed by induction on the structure of this transition.

- S-Out: we have $\bar{x}y.P \xrightarrow{\bar{x}y} P$; we immediately conclude that $x, y \in \mathsf{fn}(\bar{x}y.P)$.
- S-Par-L: we have $P \mid Q \xrightarrow{\bar{x}y} P' \mid Q$ and we obtain that $P \xrightarrow{\bar{x}y} P'$.
  By applying the inductive hypothesis on $P \xrightarrow{\bar{x}y} P'$ we obtain that $x, y \in \mathsf{fn}(P)$; since $\mathsf{fn}(P \mid Q) = \mathsf{fn}(P) \cup \mathsf{fn}(Q)$, we conclude that $x, y \in \mathsf{fn}(P \mid Q)$.
- S-Par-R: analogous to the previous case.
- S-Res: we have $(\nu z)P \xrightarrow{\bar{x}y} (\nu z)P'$ and we obtain that $P \xrightarrow{\bar{x}y} P'$ and $z \neq x, y$.
  By applying the inductive hypothesis on $P \xrightarrow{\bar{x}y} P'$ we obtain that $x, y \in \mathsf{fn}(P)$; since $\mathsf{fn}((\nu z)P) = \mathsf{fn}(P) \setminus \{z\}$ and $z \neq x, y$, we conclude that $x, y \in \mathsf{fn}((\nu z)P)$.

□

Lemma 2.2. *If $P \xrightarrow{x(y)} P'$, then $x \in$ fn(P).*

Proof. As in the previous lemma, we assume $P \xrightarrow{x(y)} P'$ and proceed by induction on the structure of this transition.

- S-In: we have $x(z).P \xrightarrow{x(y)} P\{y/z\}$; we conclude that $x \in \mathsf{fn}(x(z).P)$.

The other cases (S-Par-L, S-Par-R, S-Res) are analogous to the previous lemma.

□

Lemma 2.3. *If $P \xrightarrow{\bar{x}(z)} P'$, then $x \in$ fn(P) and $z \in$ bn(P).*

Proof. As in the previous lemma, we assume $P \xrightarrow{\bar{x}(z)} P'$ and proceed by induction on the structure of this transition.

- S-Open: we have that $(\nu z)P \xrightarrow{\bar{x}(z)} P'$ and we obtain that $x \neq z$ and $P \xrightarrow{\bar{x}z} P'$. By applying Lemma 2.1 on the last transition we obtain that $x, z \in \mathsf{fn}(P)$. Since $x \neq z$ we can conclude that $x \in \mathsf{fn}((\nu z)P)$ and $z \in \mathsf{bn}((\nu z)P)$.

The other cases (S-Par-L, S-Par-R, S-Res) are analogous to the previous lemma.

□

Lemma 2.4. *If $P \xrightarrow{\alpha} P'$, $x \notin$ n(α) and $x \notin$ fn(P), then $x \notin$ fn(P′).*

Proof. Let $P \xrightarrow{\alpha} P'$, with $x \notin \mathsf{n}(\alpha)$ and $x \notin \mathsf{fn}(P)$. We proceed by induction on the structure of the transition.

- S-In: we have $z(w).P \xrightarrow{z(y)} P\{y/w\}$, with $x \neq y, z$ and $x \notin \mathsf{fn}(z(w).P)$.
  $x \notin \mathsf{fn}(z(w).P)$ implies that $x \notin (\mathsf{fn}(P) \setminus \{w\})$. Moreover, since $\mathsf{fn}(P\{y/w\}) \subseteq (\mathsf{fn}(P) \setminus \{w\}) \cup \{y\}$, we obtain that $x \notin \mathsf{fn}(P\{y/w\})$.

- S-Out: we have $\bar{z}y.P \xrightarrow{\bar{z}y} P$, with $x \neq y, z$ and $x \notin \mathsf{fn}(\bar{z}y.P)$. The last assertion implies that $x \notin \mathsf{fn}(P)$.

- S-Par-L: we have $P \mid Q \xrightarrow{\alpha} P' \mid Q$, with $x \notin \mathsf{n}(\alpha)$ and $x \notin \mathsf{fn}(P \mid Q)$; we obtain that $P \xrightarrow{\alpha} P'$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset$.

  $x \notin \mathsf{fn}(P \mid Q)$ implies that $x \notin \mathsf{fn}(P)$ and $x \notin \mathsf{fn}(Q)$. Thus, by applying the inductive hypothesis on the set $\{P \xrightarrow{\alpha} P',\ x \notin \mathsf{n}(\alpha), x \notin \mathsf{fn}(P)\}$ we obtain that $x \notin \mathsf{fn}(P')$. Since $x \notin \mathsf{fn}(P')$ and $x \notin \mathsf{fn}(Q)$, we conclude that $x \notin \mathsf{fn}(P' \mid Q)$.

- S-Par-R: analogous to the previous case.

- S-Com-L: we have $P \mid Q \xrightarrow{\tau} P' \mid Q'$, with $x \notin \mathsf{fn}(P \mid Q)$; we obtain that $P \xrightarrow{\bar{z}y} P'$ and $Q \xrightarrow{z(y)} Q'$.

  $x \notin \mathsf{fn}(P \mid Q)$ implies that $x \notin \mathsf{fn}(P)$ and $x \notin \mathsf{fn}(Q)$. According to Lemma 2.1, $P \xrightarrow{\bar{z}y} P'$ implies that $y, z \in \mathsf{fn}(P)$, and since x is not free in P we conclude that $x \neq y, z$. By applying the inductive hypothesis on the sets $\{P \xrightarrow{\bar{z}y} P',\ x \neq y, z,\ x \notin \mathsf{fn}(P)\}$ and $\{Q \xrightarrow{z(y)} Q',\ x \neq y, z,\ x \notin \mathsf{fn}(Q)\}$ we obtain that $x \notin \mathsf{fn}(P')$ and $x \notin \mathsf{fn}(Q')$. Hence, we conclude that $x \notin \mathsf{fn}(P' \mid Q')$.

- S-Com-R: analogous to the previous case.

- S-Res: we have $(\nu z)P \xrightarrow{\alpha} (\nu z)P'$, with $x \notin \mathsf{n}(\alpha)$ and $x \notin \mathsf{fn}((\nu z)P)$; we obtain that $P \xrightarrow{\alpha} P'$ and $z \notin \mathsf{n}(\alpha)$.

  If $x = z$, then x cannot occur free in $(\nu x)P'$ (in case x occurred free in P', it would result being bound in $(\nu x)P'$).

  If $x \neq z$, since $\mathsf{fn}(P) \subseteq \mathsf{fn}((\nu z)P) \cup \{z\}$ and $x \notin \mathsf{fn}((\nu z)P) \cup \{z\}$, we obtain that $x \notin \mathsf{fn}(P)$. Thus, by applying the inductive hypothesis on the set $\{P \xrightarrow{\alpha} P', x \notin \mathsf{n}(\alpha), x \notin \mathsf{fn}(P)\}$ we obtain that $x \notin \mathsf{fn}(P')$.
  Since $\mathsf{fn}((\nu z)P') \subseteq \mathsf{fn}(P')$ we conclude that $x \notin \mathsf{fn}((\nu z)P')$.

- S-Open: we have $(\nu z)P \xrightarrow{\bar{y}(z)} P'$, with $x \neq y, z$ and $x \notin \mathsf{fn}((\nu z)P)$; we obtain that $P \xrightarrow{\bar{y}z} P'$ and $y \neq z$.

  Since $\mathsf{fn}(P) \subseteq \mathsf{fn}((\nu z)P) \cup \{z\}$ and $x \notin \mathsf{fn}((\nu z)P) \cup \{z\}$, we deduce that $x \notin \mathsf{fn}(P)$. Thus, by applying the inductive hypothesis on the set $\{P \xrightarrow{\bar{y}z} P', x \neq y, z,\ x \notin \mathsf{fn}(P)\}$ we obtain that $x \notin \mathsf{fn}(P')$.

- S-Close-L: we have $P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')$, with $x \notin \mathsf{fn}(P \mid Q)$; we obtain that $P \xrightarrow{\bar{y}(z)} P'$, $Q \xrightarrow{y(z)} Q'$ and $z \notin \mathsf{fn}(Q)$.

  If $x = z$, then x cannot occur free in $(\nu x)(P' \mid Q')$.

  Let then $x \neq z$. $x \notin \mathsf{fn}(P \mid Q)$ implies $x \notin \mathsf{fn}(P)$ and $x \notin \mathsf{fn}(Q)$. According to Lemma 2.2, $Q \xrightarrow{y(z)} Q'$ implies that $y \in \mathsf{fn}(Q)$; x does not occur free in Q, so we obtain that $x \neq y$. By applying the inductive hypothesis on the sets $\{P \xrightarrow{\bar{y}(z)} P', x \neq y, z,\ x \notin\ \mathsf{fn}(P)\}$ and $\{Q \xrightarrow{y(z)} Q', x \neq y, z,\ x \notin \mathsf{fn}(Q)\}$ we obtain that $x \notin \mathsf{fn}(P')$ and $x \notin \mathsf{fn}(Q')$. Hence, we infer that $x \notin \mathsf{fn}(P' \mid Q')$; finally, since $\mathsf{fn}((\nu z)(P' \mid Q')) \subseteq \mathsf{fn}(P' \mid Q')$, we conclude that $x \notin \mathsf{fn}((\nu z)(P' \mid Q'))$.

- S-Close-R: analogous to the previous case.

$\square$

Lemma 2.5. *If $P \equiv P'$, then $x \in \mathsf{fn}(P) \Leftrightarrow x \in \mathsf{fn}(P')$.*

Proof. Let $P \equiv P'$; we proceed by induction on the structure of $P \equiv P'$.

- Par-Unit: we have $P \mid \mathbf{0} \equiv P$. The conclusion follows from the fact that $\mathsf{fn}(P \mid \mathbf{0}) = \mathsf{fn}(P) \cup \mathsf{fn}(\mathbf{0}) = \mathsf{fn}(P)$.

- Par-Comm: we have $P \mid Q \equiv Q \mid P$. The conclusion follows from the fact that $\mathsf{fn}(P \mid Q) = \mathsf{fn}(P) \cup \mathsf{fn}(Q) = \mathsf{fn}(Q \mid P)$.

- Par-Assoc: we have $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$; analogous to the previous case.

- Sc-Ext-Zero: we have $(\nu x)\,\mathbf{0} \equiv \mathbf{0}$. Both processes have no free occurrences of variables.

- Sc-Ext-Par: we have $(\nu z)P \mid Q \equiv (\nu z)(P \mid Q)$, with $z \notin \mathsf{fn}(Q)$.

  - Let $x \in \mathsf{fn}((\nu z)P \mid Q)$: then either $x \in \mathsf{fn}((\nu z)P)$ or $x \in \mathsf{fn}(Q)$ holds. In the first case, $x \in \mathsf{fn}(P)$ and $x \neq z$, hence $x \in \mathsf{fn}(P \mid Q)$ and then $x \in \mathsf{fn}((\nu z)(P \mid Q))$. In the second case, $x \neq z$ because x occurs freely in Q and z does not; hence $x \in \mathsf{fn}(P \mid Q)$ and then $x \in \mathsf{fn}((\nu z)(P \mid Q))$.
  - Let $x \in \mathsf{fn}((\nu z)(P \mid Q))$: then $x \in \mathrm{fn}(P \mid Q)$, hence either $x \in \mathsf{fn}(P)$ or $x \in \mathsf{fn}(Q)$, and $x \neq z$. If $x \in \mathsf{fn}(P)$, then we obtain that $x \in \mathsf{fn}((\nu z)P)$, since $x \neq z$. As a result, in both cases we conclude that $x \in \mathsf{fn}((\nu z)P \mid Q)$.

- Sc-Ext-Res: we have $(\nu y)(\nu z)P \equiv (\nu z)(\nu y)P$. The conclusion follows from the fact that $\mathsf{fn}((\nu y)(\nu z)P) = \mathsf{fn}((\nu z)P)\backslash\{y\} = \mathsf{fn}(P)\backslash\{y, z\} =$ $= \mathsf{fn}((\nu y)P)\backslash\{z\} = \mathsf{fn}((\nu z)(\nu y)P)$.

- C-In: we have $z(y).P \equiv z(y).Q$, given that $P \equiv Q$.

  If $x \in \mathsf{fn}(z(y).P)$, then either $x = z$ (and we conclude immediately) or $x \neq y$ and $x \in \mathsf{fn}(P)$. In this case, by applying the inductive hypothesis on the congruence $P \equiv Q$ we obtain that $x \in \mathsf{fn}(Q)$; since $x \neq y$, we conclude that $x \in \mathsf{fn}(z(y).Q)$. The other implication is analogous.

- C-Out: we have $\bar{z}y.P \equiv \bar{z}y.Q$, given that $P \equiv Q$.

  If $x \in \mathsf{fn}(\bar{z}y.P)$, then either $x = z$, $x = y$ or $x \in \mathsf{fn}(P)$. In the first two cases we conclude immediately; in the last case, we apply the inductive hypothesis and conclude. The other implication is analogous.

- C-Par: we have $P \mid Q \equiv P' \mid Q$, given that $P \equiv P'$.

  Let $x \in \mathsf{fn}(P \mid Q)$. If $x \in \mathsf{fn}(P)$, then we apply the inductive hypothesis and conclude; if $x \in \mathsf{fn}(Q)$ we conclude immediately. The other implication is analogous.

- C-Res: we have $(\nu z)P \equiv (\nu z)Q$, given that $P \equiv Q'$.

  If $x \in \mathsf{fn}((\nu z)P)$, then $x \in \mathsf{fn}(P)$ and $x \neq z$; we apply the inductive hypothesis and conclude. The other implication is analogous.

- C-Ref: we have $P \equiv P$. The proof is immediate.

- C-Sym: we have $Q \equiv P$, given that $P \equiv Q$. By applying the inductive hypothesis on the congruence $P \equiv Q$ we obtain that $x \in \mathsf{fn}(P) \Leftrightarrow x \in \mathsf{fn}(Q)$.

- C-Trans: we have $P \equiv R$, given that $P \equiv Q$ and $Q \equiv R$.

  If $x \in \mathsf{fn}(P)$, then $x \in \mathsf{fn}(Q)$ by inductive hypothesis and $x \in \mathsf{fn}(R)$ by inductive hypothesis. The other implication is analogous.

$\square$

We state the first of two main lemmas needed for the proof of Theorem 2. The proof is a long induction on the structure of the congruence $P \equiv Q$ given as hypothesis: we present the most significant steps, leaving the rest of the proof to Appendix B.

Lemma 2.6. *If $P \equiv Q$ and $P \xrightarrow{\alpha} P'$, then there exists a process Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \equiv Q'$.*

Remark. The proof is carried out by induction over the structure of the congruence $P \equiv P'$. It requires assuming that the left-hand side process P makes some transition and proving that the right-hand side process P' makes an appropriate transition. However, when considering the case where this congruence arises from the C-Sym rule, the roles of the left and right-hand side processes are reversed: the assumption involves the right-hand side process and the goal involves the left-hand side process. In such situations, in order to streamline the proof structure and automatically address the C-Sym case, the two symmetric assertions can be proved concurrently, as displayed in the following commuting diagrams. In these diagrams, solid arrows and symbols denote universal quantification, while dashed arrows and symbols denote existential quantification.

$$\begin{tikzcd}
P \arrow[r, phantom, "\equiv"] \arrow[d, "\alpha"] & Q \arrow[d, dashed, "\alpha"] & & P \arrow[r, phantom, "\equiv"] \arrow[d, dashed, "\alpha"] & Q \arrow[d, "\alpha"] \\
P' \arrow[r, phantom, "\equiv"] & Q' & & P' \arrow[r, phantom, "\equiv"] & Q'
\end{tikzcd}$$

Figure 2.6: Graphical representation of Lemma 2.6 statement.

Proof. Let $P \equiv Q$. As explained in the remark above, we proceed by induction on the structure of $P \equiv Q$; for each case, we first assume $P \xrightarrow{\alpha} P'$, aiming to obtain a transition of the process Q, and then we assume $Q \xrightarrow{\alpha} Q'$, aiming to obtain a transition of P.

- Par-Assoc: we have $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$.

  Let $P \mid (Q \mid R) \xrightarrow{\alpha} S$, denoted as (T1). We perform inversion on this transition, obtaining the following subcases:

- S-PAR-R: (T1) can be rewritten as $P \mid (Q \mid R) \xrightarrow{\alpha} P \mid S'$, and we have that $(Q \mid R) \xrightarrow{\alpha} S'$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(P) = \emptyset$. We perform inversion on the transition $(Q \mid R) \xrightarrow{\alpha} S'$, obtaining different subcases.
  - S-CLOSE-L : (T1) is rewritten as $P \mid (Q \mid R) \xrightarrow{\tau} P \mid ((\nu z)(Q' \mid R'))$, and we have that $z \notin \mathsf{fn}(R)$, $Q \xrightarrow{\bar{x}(z)} Q'$ and $R \xrightarrow{x(z)} R'$.
    Applying Lemma 2.3, $Q \xrightarrow{\bar{x}(z)} Q'$ implies that $z \in \mathsf{bn}(Q)$; for the variable convention, we can assume that z is not free in P. Hence, we can apply a S-PAR-R rule, obtaining $P \mid Q \xrightarrow{\bar{x}(z)} P \mid Q'$.
    Then, since z is not free in R, we can apply a S-CLOSE-L rule, obtaining $(P \mid Q) \mid R \xrightarrow{\tau} (\nu z)((P \mid Q') \mid R')$.
    Finally, we show that $P \mid ((\nu z)(Q' \mid R')) \equiv (\nu z)((P \mid Q') \mid R')$ through the following congruences:

$$\textsc{Par-Comm}\ \frac{}{P \mid ((\nu z)(Q' \mid R')) \equiv ((\nu z)(Q' \mid R')) \mid P}\ (C1)$$

$$\textsc{Sc-Ext-Par}\ \frac{z \notin \mathsf{fn}(P)}{((\nu z)(Q' \mid R')) \mid P \equiv (\nu z)((Q' \mid R') \mid P)}\ (C2)$$

$$\textsc{C-Res}\ \frac{\textsc{Par Monoid Axioms}\ \frac{}{(Q' \mid R') \mid P \equiv (P \mid Q') \mid R'}}{(\nu z)((Q' \mid R') \mid P) \equiv (\nu z)((P \mid Q') \mid R')}\ (C3)$$

$$\textsc{C-Trans}\ \frac{\textsc{C-Trans}\ \frac{(C1) \quad (C2)}{(C4)} \qquad (C3)}{P \mid ((\nu z)(Q' \mid R')) \equiv (\nu z)((P \mid Q') \mid R')}$$

  - ...
- ...

- SC-EXT-PAR: we have $(\nu x)P \mid Q \equiv (\nu x)(P \mid Q)$, with $x \notin \mathsf{fn}(Q)$.

  First, let $(\nu x)P \mid Q \xrightarrow{\alpha} R$, denoted as (T1). We perform inversion on this transition, obtaining the following subcases.

  - S-PAR-R : (T1) can be rewritten as $(\nu x)P \mid Q \xrightarrow{\alpha} (\nu x)P \mid Q'$, and we have that $Q \xrightarrow{\alpha} Q'$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}((\nu x)P) = \emptyset$.
    For the variable convention we can assume that the bound name x does not occur free or bound in the action α: thus we have both the conditions $x \notin \mathsf{bn}(\alpha)$ and $x \notin \mathsf{n}(\alpha)$.
    Since $\mathsf{fn}(P) \subseteq \mathsf{fn}((\nu x)P) \cup \{x\}$, $\mathsf{bn}(\alpha) \cap \mathsf{fn}((\nu x)P) = \emptyset$ and $x \notin \mathsf{bn}(\alpha)$, we have that $\mathsf{bn}(\alpha) \cap \mathsf{fn}(P) = \emptyset$. Thus, we can apply a S-PAR-R rule, obtaining that $P \mid Q \xrightarrow{\alpha} P \mid Q'$. Then, since $x \notin \mathsf{n}(\alpha)$, we can apply a S-RES rule, obtaining that $(\nu x)(P \mid Q) \xrightarrow{\alpha} (\nu x)(P \mid Q')$.
    Finally, since $Q \xrightarrow{\alpha} Q'$, $x \notin \mathsf{fn}(Q)$ and $x \notin \mathsf{n}(\alpha)$, through Lemma 2.4 we obtain that $x \notin \mathsf{fn}(Q')$; hence, we conclude by observing that the process $(\nu x)(P \mid Q')$ is congruent to the process $(\nu x)P \mid Q'$ through a SC-EXT-PAR rule.
  - ...

Conversely, let $(\nu x)(P \mid Q) \xrightarrow{\alpha} R$, denoted as (T2). By inversion on this transition we obtain two subcases:

- S-Res: we have that $x \notin \mathsf{n}(\alpha)$ and $P \mid Q \xrightarrow{\alpha} S$. We perform inversion on this transition, obtaining the following subcases.
  - S-Com-L : (T2) can be rewritten as $(\nu x)(P \mid Q) \xrightarrow{\tau} (\nu x)(P' \mid Q')$, and we have that $P \xrightarrow{\bar{z}w} P'$ and $Q \xrightarrow{z(w)} Q'$.
    Since $Q \xrightarrow{z(w)} Q'$, by applying Lemma 2.2 we deduce that $z \in \mathsf{fn}(Q)$; however, by hypothesis $x \notin \mathsf{fn}(Q)$, thus we conclude that $x \neq z$. We have two cases, depending on whether $x = w$ or $x \neq w$.
    If $x \neq w$, we can apply the S-Res rule to obtain the transition $(\nu x)P \xrightarrow{\bar{z}w} (\nu x)P'$. Then, through a S-Com-L we obtain that
    $(\nu x)P \mid Q \xrightarrow{\tau} (\nu x)P' \mid Q'$. Since $Q \xrightarrow{z(w)} Q'$, $x \notin \mathsf{fn}(Q)$ and $x \neq z, w$, through Lemma 2.4 we deduce that $x \notin \mathsf{fn}(Q')$; finally, through a Sc-Ext-Par rule we obtain that the process $(\nu x)P' \mid Q'$ is congruent to the process $(\nu x)(P' \mid Q')$.
    If $x = w$, we can apply the S-Open rule to obtain the transition $(\nu x)P \xrightarrow{\bar{z}(x)} P'$. Then, through a S-Close-L we get that
    $(\nu x)P \mid Q \xrightarrow{\tau} (\nu x)(P' \mid Q')$, hence reaching our goal.
  - S-Close-L : (T2) is rewritten as $(\nu x)(P \mid Q) \xrightarrow{\tau} (\nu x)((\nu w)(P' \mid Q'))$, and we have that $P \xrightarrow{\bar{z}(w)} P'$, $Q \xrightarrow{z(w)} Q'$ and $w \notin \mathsf{fn}(Q)$.
    Since $Q \xrightarrow{z(w)} Q'$, by applying Lemma 2.2 we have that $z \in \mathsf{fn}(Q)$; however, by hypothesis $x \notin \mathsf{fn}(Q)$, thus we conclude that $x \neq z$. On the other hand, since $P \xrightarrow{\bar{z}(w)} P'$, by applying Lemma 2.3 we have that $w \in \mathsf{bn}(P)$; this implies that either a restriction (νw) or an input prefix $y(w)$ occurs in P, binding w in P. Considering that the restriction (νx) in the process $(\nu x)(P \mid Q)$ is outermost, we derive that the occurrences of the restriction (νx) and the construct which binds w in P are distinct; hence, up to α-renaming, we can assume that $x \neq w$. For this reason we can apply the S-Res rule to obtain the transition $(\nu x)P \xrightarrow{\bar{z}(w)} (\nu x)P'$. Finally, through a S-Close-L we get that $(\nu x)P \mid Q \xrightarrow{\tau} (\nu w)((\nu x)P' \mid Q')$.
    Since $Q \xrightarrow{z(w)} Q'$, $x \notin \mathsf{fn}(Q)$ and $x \neq z, w$, through Lemma 2.4 we have that $x \notin \mathsf{fn}(Q')$; thus, through a Sc-Ext-Par rule we obtain that $(\nu x)P' \mid Q'$ is congruent to $(\nu x)(P' \mid Q')$. Through a C-Res rule we obtain that $(\nu w)((\nu x)P' \mid Q')$ is congruent to $(\nu w)((\nu x)(P' \mid Q'))$; the latter process is congruent to $(\nu x)((\nu w)(P' \mid Q'))$ via Sc-Ext-Res, hence achieving our goal.
  - ...
- S-Open : (T2) can be rewritten as $(\nu x)(P \mid Q) \xrightarrow{\bar{z}(x)} R$, and we have that $z \neq x$ and $(P \mid Q) \xrightarrow{\bar{z}x} R$. We perform inversion on this transition through the S-Par-L rule, as the S-Par-R would lead to a contradiction: namely, we would obtain that $Q \xrightarrow{\bar{z}x} Q'$, thus by applying Lemma 2.1 we would deduce $\{z, x\} \subseteq \mathsf{fn}(Q)$, which is absurd due to the hypothesis

$x \notin \mathsf{fn}(Q)$. Hence, (T1) can be rewritten as $(\nu x)(P \mid Q) \xrightarrow{\bar{z}(x)} P' \mid Q$, and we have that $P \xrightarrow{\bar{z}x} P'$.

Through S-Open we obtain that $(\nu x)P \xrightarrow{\bar{z}(x)} P'$; since $x \notin \mathsf{fn}(Q)$ we can apply the S-Par-L rule and obtain that $(\nu x)P \mid Q \xrightarrow{\bar{z}(x)} P' \mid Q$, hence achieving our goal.

- C-Par : we have $P \mid Q \equiv P' \mid Q$, given that $P \equiv P'$.

  First, let $P \mid Q \xrightarrow{\alpha} R$, denoted as (T1). We perform inversion on this transition, obtaining the following subcases.

  – S-Par-R: (T1) can be rewritten as $P \mid Q \xrightarrow{\alpha} P \mid Q'$, and we have $Q \xrightarrow{\alpha} Q'$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(P) = \emptyset$.

  Since $P \equiv P'$ and $x \in \mathsf{fn}(P)$, by applying Lemma 2.5 we deduce that $x \in \mathsf{fn}(P')$. As a consequence, since $\mathsf{bn}(\alpha) \cap \mathsf{fn}(P) = \emptyset$, we have that $\mathsf{bn}(\alpha) \cap \mathsf{fn}(P') = \emptyset$. Thus, we can apply the S-Par-R rule and obtain that $P' \mid Q \xrightarrow{\alpha} P' \mid Q'$. The process $P \mid Q'$ is congruent to the process $P' \mid Q'$ through C-Par, hence achieving our goal.

  – ...

- ...

$\square$

We state and prove the last lemma needed for the proof of Theorem 2.

Lemma 2.7. *If $P \to Q$ then there exist three names x, y and z, a finite (possibly empty) set of names $w_1, ..., w_n$ and three processes R_1, R_2 and S such that*
$P \equiv (\nu w_1)...(\nu w_n)(\,(\bar{x}y.R_1 \mid x(z).R_2) \mid S\,)$ and
$Q \equiv (\nu w_1)...(\nu w_n)(\,(R_1 \mid R_2\{y/z\}) \mid S\,)$.

Proof. Let $P \to Q$; we proceed by induction on the structure of this reduction.

- R-Com: we have $\bar{x}y.R_1 \mid x(z).R_2 \;\to\; R_1 \mid R_2\{y/z\}$.

  With an empty set of names and setting $S := \mathbf{0}$, we obtain through Par-Unit that $P \equiv (\bar{x}y.R_1 \mid x(z).R_2) \mid S$ and $Q \equiv (R_1 \mid R_2\{y/z\}) \mid S$, thus reaching the conclusion.

- R-Par: we have $P \mid R \to Q \mid R$, given that $P \to Q$.

  By applying the inductive hypothesis on the reduction $P \to Q$ we obtain that there exist names $x, y, z, w_1, ..., w_n$ and processes R_1, R_2, S such that
  $P \equiv (\overline{\nu w})((\bar{x}y.R_1 \mid x(z).R_2) \mid S)$ and $Q \equiv (\overline{\nu w})((R_1 \mid R_2\{y/z\}) \mid S)$, with $(\overline{\nu w}) := (\nu w_1)...(\nu w_n)$.
  Through C-Par we obtain that $P \mid R \equiv ((\overline{\nu w})((\bar{x}y.R_1 \mid x(z).R_2) \mid S)) \mid R$ and $Q \mid R \equiv ((\overline{\nu w})((R_1 \mid R_2\{y/z\}) \mid S)) \mid R$.
  We can assume that, up to α-renaming, the names $w_1, ..., w_n$ do not occur free in R: so, by applying the Sc-Ext-Par rule n times, we obtain that $((\overline{\nu w})((\bar{x}y.R_1 \mid x(z).R_2) \mid S)) \mid R \equiv (\overline{\nu w})(((\bar{x}y.R_1 \mid x(z).R_2) \mid S) \mid R)$ and $((\overline{\nu w})((R_1 \mid R_2\{y/z\}) \mid S)) \mid R \equiv (\overline{\nu w})(((R_1 \mid R_2\{y/z\}) \mid S) \mid R)$.

Thanks to a combination of monoid axioms for parallel composition, followed by n applications of the C-Res rule, we obtain the following congruences:

$$(\overline{\nu w})(((\bar{x}y.R_1 \mid x(z).R_2) \mid S) \mid R\,) \equiv (\overline{\nu w})(\,(\bar{x}y.R_1 \mid x(z).R_2) \mid (S \mid R)\,);$$
$$(\overline{\nu w})(((R_1 \mid R_2\{y/z\}) \mid S) \mid R\,) \equiv (\overline{\nu w})(\,(R_1 \mid R_2\{y/z\}) \mid (S \mid R)\,).$$

Finally, by applying the C-Trans rule twice we obtain that the processes $P \mid R$ and $Q \mid R$ are equivalent to the desired processes.

- R-Res: we have $(\nu v)P \to (\nu v)Q$, given that $P \to Q$.

  By applying the inductive hypothesis on the reduction $P \to Q$ we obtain that there exist names $x, y, z, w_1, ..., w_n$ and processes R_1, R_2, S such that $P \equiv (\overline{\nu w})((\bar{x}y.R_1 \mid x(z).R_2) \mid S)$ and $Q \equiv (\overline{\nu w})((R_1 \mid R_2\{y/z\}) \mid S)$. Through a C-Res rule we obtain that $(\nu v)P \equiv (\nu v)(\overline{\nu w})((\bar{x}y.R_1 \mid x(z).R_2) \mid S)$ and $(\nu v)Q \equiv (\nu v)(\overline{\nu w})((R_1 \mid R_2\{y/z\}) \mid S)$, thus reaching the conclusion.

- R-Struct: we have $P \to Q$, given that $P' \to Q'$, $P \equiv P'$ and $Q' \equiv Q$.

  By applying the inductive hypothesis on the reduction $P' \to Q'$ we obtain that there exist names $x, y, z, w_1, ..., w_n$ and processes R_1, R_2, S such that $P' \equiv (\overline{\nu w})((\bar{x}y.R_1 \mid x(z).R_2) \mid S)$ and $Q' \equiv (\overline{\nu w})((R_1 \mid R_2\{y/z\}) \mid S)$. By transitivity, $P \equiv P'$ and $P' \equiv (\overline{\nu w})((\bar{x}y.R_1 \mid x(z).R_2) \mid S)$ imply that $P \equiv (\overline{\nu w})((\bar{x}y.R_1 \mid x(z).R_2) \mid S)$. Since $Q' \equiv Q$ implies $Q \equiv Q'$, we proceed analogously for Q.

□

Finally, we prove Theorem 2. We recall its statement:

Theorem 2. *$P \to Q$ implies the existence of a Q' such that $P \xrightarrow{\tau} Q'$ and $Q \equiv Q'$.*

Proof. Let $P \to Q$.
By applying Lemma 2.7, there exist names $x, y, z, w_1, ..., w_n$ and processes R_1, R_2, S such that $P \equiv P''$ and $Q \equiv Q''$, with $P'' := (\nu w_1)...(\nu w_n)((\bar{x}y.R_1 \mid x(z).R_2) \mid S)$ and $Q'' := (\nu w_1)...(\nu w_n)((R_1 \mid R_2\{y/z\}) \mid S)$.
By applying a S-Com-L rule, followed by a S-Par-L rule and n S-Res rules, we obtain that $P'' \xrightarrow{\tau} Q''$: thus, by applying Lemma 2.6 on the set $\{P'' \equiv P,\ P'' \xrightarrow{\tau} Q''\}$ we obtain that there exists a process Q' such that $P \xrightarrow{\tau} Q'$ and $Q'' \equiv Q'$. Through a C-Trans rule we obtain that $Q \equiv Q'$, hence the conclusion. □

The following diagram illustrates the congruences and transitions used in the proof.

$$\begin{tikzcd}
P \arrow[d] & & & & P'' \arrow[d, "\tau"] & \equiv & P \arrow[d, dashed, "\tau"] \\
Q & \Rightarrow & Q & \equiv & Q'' & \equiv & Q'
\end{tikzcd}$$

Figure 2.7: Graphical representation of Theorem 2 proof.

Chapter 3

Beluga Formalization

Beluga ([19], [20], [21], [22]) is a functional programming language and proof environment developed by the Complogic group at McGill University in Montreal, Canada, under the leadership of Professor Brigitte Pientka. It offers a two-level infrastructure to formalize and reason about deductive systems. The data level deploys the logical framework LF [8], which enables the definition of type families to represent data of the object language. The computation level supports contexts and contextual objects, which are employed to reason about LF type families and prove properties about them.

Beluga implements higher-order abstract syntax [18], a technique which allows the representation of binding constructs from the object language through binders of the meta-language. This approach effectively exempts the user from having to handle substitutions and renamings directly. Proofs are represented by recursive programs, according to the Curry-Howard isomorphism [11]; in order to ensure the correctness of proofs, Beluga includes a totality checker, which verifies that all cases are covered and recursive calls are made on structurally smaller arguments. Moreover, Beluga supports the use of computation holes [14] to represent subgoals in the proof terms; it interactively assists the user by inferring the type of computation holes and by attempting to automatically split variables for case analysis upon request.

The current chapter outlines the Beluga formalization of the definitions and proofs introduced in Chapter 2. Following the structure of the informal presentation, the first sections focus on the encoding of the syntax and semantics of the π-calculus; this includes a gradual explanation of the Beluga constructs and features used. It is important to recall that the Beluga encoding will employ the late LTS semantics, instead of the early semantics previously adopted. The following sections detail the key steps involving the formalization of the theorems about semantics equivalence; we also discuss the challenges encountered during the encoding process and the corresponding solutions devised.

3.1 Syntax

To begin with, we define the sets of names and processes. In Beluga, data of the object language are introduced by declaring new types in the logical framework LF [8] along with their constructors [22].
The set of names is a countably infinite set without any other inherent property

or constraint. The following declaration defines an LF type names which does not contain any predefined constant or have any constructor:

```
LF names: type =
;
```

The type names can be thought of as initially empty, with the possibility of having any number of distinct names added at need.

The next declaration defines an LF type proc along with its constructors p_zero, p_in, p_out, p_par and p_res:

```
LF proc: type =
  | p_zero: proc
  | p_in: names → (names → proc) → proc
  | p_out: names → names → proc → proc
  | p_par: proc → proc → proc
  | p_res: (names → proc) → proc
;
--infix p_par 11 left.
```

Figure 3.1: Encoding of the set of processes.

The constructor p_zero is a constant of type proc and represents the process **0**. The constructor p_out defines an expression of type proc which takes two terms of type names and a term of type proc as arguments: for example the expression p_out X Y P, where X and Y have type names and P has type proc, is itself a term of type proc representing the output prefix $\bar{x}y.P$. Similarly, the constructor p_par takes two terms of type proc as arguments and represents parallel composition $P \mid Q$.
The constructor p_in takes a term of type names and a term of type (names → proc) as arguments, with the latter being the type of higher-order functions from names to proc; higher-order functions can be explicitly described in Beluga by the syntax \x.(F x), where x is the argument of the function and F is the body of the function. Thus, the input prefix $x(y).P$ is represented by the expression p_in X P, where P is a variable of the functional type (names → proc). By employing higher-order abstract syntax [18], we are able to represent the object-level binder y through a meta-language binder, namely the implicit argument of the function P; one of the key benefits of this technique is that the meta-language automatically implements α-renaming for binders and capture-avoiding substitutions [20]. As a consequence, the variable conventions presented in section 2.1.1 are achieved without being explicitly expressed by the user; this also holds for the side conditions for semantics rules and for most of the lemmas about free and bound variables in processes, as it will be discussed in the next sections.
In the same way, the constructor p_res takes a higher-order function P from names to proc as an argument, and represents the restriction $(\nu x)P$ by the expression p_res P. The last line following the LF type proc declaration makes the p_par constructor infix.

Given a language encoding, it is important to show fidelity of the representation to the original model. This is achieved by estabilishing the adequacy of the representation, i.e. proving that there is a correspondence, which preserves composition, between the informal definitions of syntax and semantics and the relative types and judgements encoding them. In our context, adequacy of syntax consists in building an encoding function between the set of α-equivalence classes of processes in Proc and the elements of type proc and proving that it is a compositional bijection. Formal definition and proof of adequacy is beyond the scope of this thesis, but we refer to Honsell et al. [10] for further details. Based on this result, in the following sections we will often make no distinction between names or processes of the π-calculus and their representation in Beluga.

3.2 Semantics

In this section we present the Beluga encoding for the two different operational semantics. In alignment with the structure outlined in Section 2.2, we first introduce the encodings of structural congruence and reduction for the reduction semantics, and then introduce the encodings of actions and transition relation for the LTS semantics.

3.2.1 Reduction Semantics

Structural Congruence

In Beluga, predicates are encoded by LF type families, as clarified later. The congruence relation is defined through the following declaration:

```
LF cong: proc → proc → type =
% Abelian Monoid Laws for Parallel Composition
  | par_assoc: cong (P p_par (Q p_par R)) ((P p_par Q) p_par R)
  | par_unit: cong (P p_par p_zero) P
  | par_comm: cong (P p_par Q) (Q p_par P)
% Scope Extension Laws
  | sc_ext_zero: cong (p_res (\x.p_zero)) p_zero
  | sc_ext_par: cong ((p_res P) p_par Q) (p_res (\x.((P x) p_par Q)))
  | sc_ext_res: cong (p_res \x.(p_res \y.(P x y))) (p_res \y.(p_res \x.
    (P x y)))
% Compatibility Laws
  | c_in: ({y:names} cong (P y) (Q y)) → cong (p_in X P) (p_in X Q)
  | c_out: cong P Q → cong (p_out X Y P) (p_out X Y Q)
  | c_par: cong P P' → cong (P p_par Q) (P' p_par Q)
  | c_res: ({x:names} cong (P x) (Q x)) → cong (p_res P) (p_res Q)
% Equivalence Relation Laws
  | c_ref: cong P P
  | c_sym: cong P Q → cong Q P
  | c_trans: cong P Q → cong Q R → cong P R
;
--infix cong 11 left.
```

Figure 3.2: Encoding of the congruence relation.

The LF type cong is indexed by two terms of type proc: it means that $P \equiv Q$ holds if and only if the type `cong P Q` is inhabited. Some constructors, such as `par_comm`, do not have any explicit premise and directly assert some congruence; for example, `par_comm` does not take any argument, aside from the two implicit processes P and

Q, and denotes the congruence $P \mid Q \equiv Q \mid P$. Other constructors, such as `c_trans`, take some expressions as explicit arguments; these expressions represent premises of the corresponding inference rule. For example, `c_trans C1 C2` represents the congruence $P \equiv R$, provided that `C1` is an expression of type `cong P Q` and `C2` is an expression of type `cong Q R`.

The tokens `{y:names}` appearing in the `c_in` rule denote universal quantification: therefore, we assume as a premise that for all `y` in `names`, `cong (P y) (Q y)` holds. In this way we are stating that `P` and `Q`, which can be thought of as processes depending on a certain argument, are congruent regardless of the specific choice of the argument's name. The same holds for the `c_res` rule.
We also observe that, in the encoding of the SC-EXT-PAR rule, the process $(\nu x)P \mid Q$ is represented as `(p_res P) p_par Q`; the binder x of the left component is represented by the binder of the function `P`, which is automatically selected to be distinct from any free name occurring in `Q`. In this way, the side condition $x \notin \mathsf{fn}(Q)$ is met without the need for explicit declaration.

Reduction

The reduction relation is encoded by the LF type family `red` defined through the following declaration:

```
LF red: proc → proc → type =
  | r_com: red ((p_out X Y P) p_par (p_in X Q)) (P p_par (Q Y))
  | r_par: red P Q → red (P p_par R) (Q p_par R)
  | r_res: ({x:names} red (P x) (Q x)) → red (p_res P) (p_res Q)
  | r_str: P cong P' → red P' Q' → Q' cong Q → red P Q
;
--infix red 11 left.
```

Figure 3.3: Encoding of the reduction relation.

The encoding of the four reduction rules is straightforward. We observe that in the rule R-COM, expressed by the homonymous constructor, substitution of the name y into the process Q is performed by applying the meta-function `Q` to the name `Y` in the result process `P p_par (Q Y)`.

Adequacy of reduction semantics consists in proving that, given two processes P, Q, there exist a compositional bijection mapping $P \equiv Q$ in `P cong Q` and a compositional bijection mapping $P \to Q$ in `P red Q`. Similarly to the previous section, we refrain from presenting a formal definition of adequacy.

3.2.2 Labelled Transition System Semantics

We follow Honsell et al. [10] for the encoding of late LTS semantics in a HOAS environment. We declare two different types for bound and free actions, defining bound actions as those containing a bound name and free actions as those which do not; then, we declare two different relations for transitions via free and bound actions. The result of a free transition is a process, while the result of a bound transition is a function from names to processes: instead of explicitly stating the bound name involved in the transition, that name is the argument of the aforementioned function.

Actions

We define the two types `f_act` and `b_act` which represent free and bound actions:

```
LF f_act: type =                            LF b_act: type =
  | f_tau: f_act                              | b_in: names → b_act
  | f_out: names → names → f_act              | b_out: names → b_act
;                                           ;
```

Figure 3.4: Encoding of the set of actions.

Since the name y in the input action $x(y)$ is considered to be bound in the late semantics, the input action is encoded as a bound action of type `b_act`. The `f_out` constructor takes two arguments, representing the free output action $\bar{x}y$ as `f_out X Y`. Conversely, the `b_in` and the `b_out` constructors take one argument, representing the input action $x(y)$ and the free output action $\bar{x}(y)$ as `b_in X` and `b_out X` respectively. As previously explained, the bound name y is not explicitly instantiated in the action definition, but rather represented by the argument of a function from name to processes which constitutes the result of a bound transition.

Transition relation

We define the two type families `fstep` and `bstep` which represent free and bound transitions:

```
LF fstep: proc → f_act → proc → type =
  | fs_out: fstep (p_out X Y P) (f_out X Y) P
  | fs_par1: fstep P A P' → fstep (P p_par Q) A (P' p_par Q)
  | fs_par2: fstep Q A Q' → fstep (P p_par Q) A (P p_par Q')
  | fs_com1: fstep P (f_out X Y) P' → bstep Q (b_in X) Q'
    → fstep (P p_par Q) f_tau (P' p_par (Q' Y))
  | fs_com2: bstep P (b_in X) P' → fstep Q (f_out X Y) Q'
    → fstep (P p_par Q) f_tau ((P' Y) p_par Q')
  | fs_res: ({z:names} fstep (P z) A (P' z))
    → fstep (p_res P) A (p_res P')
  | fs_close1: bstep P (b_out X) P' → bstep Q (b_in X) Q'
    → fstep (P p_par Q) f_tau (p_res \z.((P' z) p_par (Q' z)))
  | fs_close2: bstep P (b_in X) P' → bstep Q (b_out X) Q'
    → fstep (P p_par Q) f_tau (p_res \z.((P' z) p_par (Q' z)))

and bstep: proc → b_act → (names → proc) → type =
  | bs_in: bstep (p_in X P) (b_in X) P
  | bs_par1: bstep P A P' → bstep (P p_par Q) A \x.((P' x) p_par Q)
  | bs_par2: bstep Q A Q' → bstep (P p_par Q) A \x.(P p_par (Q' x))
  | bs_res: ({z:names} bstep (P z) A (P' z))
    → bstep (p_res P) A \x.(p_res \z.(P' z x))
  | bs_open: ({z:names} fstep (P z) (f_out X z) (P' z))
    → bstep (p_res P) (b_out X) P'
;
```

Figure 3.5: Encoding of the transition relation.

The type `fstep` is indexed by the initial process, a free action and the resulting process. Conversely, the type `bstep` is indexed by the initial process, a bound action and a function from names to processes representing the resulting process. These two types are defined by mutual induction, as some of the `fstep` constructors take transitions of type `bstep` as arguments, and vice versa; in Beluga, mutual induction is achieved using the keyword `and` between the two definitions. Each of the S-Res and S-Par rules is duplicated, as they involve a generic action α which can be either free or bound. As in the R-Com rule encoding for the reduction relation, substitution of the input name in the S-Com rules is performed through a function application. Additionally, it is worth noting that side conditions of the transition rules do not need to be explicitly stated in the Beluga encoding. For instance, the S-Res rule describes the transition $(\nu x)P \xrightarrow{\alpha} (\nu x)P'$, with $P \xrightarrow{\alpha} P'$ and $x \notin \mathsf{n}(\alpha)$; the `fs_res` rule encodes the resulting transition as `f_step (p_res \x.(P x)) A (p_res \x.(P' x))`, where the binder x is represented by the argument `x` of the functions `P` and `P'`, automatically chosen to be different from the names occurring in the action `A`.

Adequacy of the LTS semantics encoding consists in demonstrating that each transition corresponds to a canonical free or bound transition of type `f_act` or `b_act`, and vice versa. The definition and proof of transition adequacy are typically less straightforward than syntax adequacy. Further details can be found in [10].

3.3 Equivalence of Reduction and LTS Semantics

In order to understand how proofs are constructed in Beluga and how the system supports the user in the proof search, we begin with an illustrative example. For this purpose, we modify the congruence relation defined in section 2.2.1 by replacing the C-Ref rule ($P \equiv P$) with the weaker C-Ref-Zero rule, which asserts that $\mathbf{0} \equiv \mathbf{0}$; with this modified congruence relation, we aim to prove that the C-Ref rule is admissable. First, we informally state and prove the example theorem.

Example. *$\forall P \in Proc$, $P \equiv P$ holds.*

Proof. Let $P \in Proc$. We proceed by induction on the structure of the process P.

- $P = \mathbf{0}$: the congruence $\mathbf{0} \equiv \mathbf{0}$ holds for the C-Ref-Zero rule.
- $P = \bar{x}y.Q$: by applying the inductive hypothesis on the process Q, we obtain that $Q \equiv Q$. Through a C-Out rule we obtain that $\bar{x}y.Q \equiv \bar{x}y.Q$, hence reaching the conclusion.
- $P = x(y).Q$: by applying the inductive hypothesis on the process Q, we obtain that $Q \equiv Q$. Through a C-In rule we obtain that $x(y).Q \equiv x(y).Q$, hence reaching the conclusion.
- $P = Q_1 \mid Q_2$: by applying the inductive hypothesis on the process Q_1, we obtain that $Q_1 \equiv Q_1$. Through a C-Par rule we obtain that $Q_1 \mid Q_2 \equiv Q_1 \mid Q_2$, hence reaching the conclusion.
- $P = (\nu x)Q$: by applying the inductive hypothesis on the process Q, we obtain that $Q \equiv Q$. Through a C-Res rule we obtain that $(\nu x).Q \equiv (\nu x).Q$, hence reaching the conclusion.

□

In Beluga, proofs are encoded by total recursive functions [22], following the Curry-Howard isomorphism [11]. Introducing an hypothesis corresponds to declaring a new variable of the appropriate type. Induction and case analysis on the structure of an object correspond to pattern matching on the corresponding variable. Application of the inductive hypothesis corresponds to recursive call of the function on a structurally smaller object.

The arguments and result of recursive functions representing theorems are not LF objects or types; instead, they are contextual objects and types, which are defined at the computation level. The context of an LF term is made of the typing assumptions of the variables occurring free in that term; an object containing free variables is referred to as an *open* object, while a *closed* object is one without free variables. A contextual object `[g ⊢ M]` consists of an LF object `M` along with its context `g`, separated by the turnstile symbol "⊢"; as the context `g` provides bindings for the free variables of the term `M`, the square brackets enclosing the contextual object denote that it is closed. On the other hand, a contextual type `[g ⊢ A]` consists of an LF type `A` paired with a context `g`; we can say that a LF object `M` has contextual type `[g ⊢ A]` in case `M:A` holds in the context `g` [19].

In the example proof, we implicitly handle open processes: specifically, in the input and restriction cases, we apply the inductive hypothesis to a process Q which contains a free variable (y in the first case, x in the second case). Consequently, in the Beluga encoding of the proof we need to deal with contextual processes and prove the result for any appropriate context. This requires a way to define contexts of a specific form and quantify over them, thus leading to the following declaration:

```
schema ctx = names;
```

This declaration ensures that the kind of context we are working with consists solely of assumptions of the form "`x:names`". In Beluga, the keyword `schema` categorizes contexts based on their structure; just like terms have their own type, contexts of the form `x:names` are classified under the schema `ctx`.

The following line replaces the `c_ref` constructor in the `cong` type declaration:

```
  | c_ref_zero: cong p_zero p_zero
```

Finally, we are ready to state the example theorem.

```
rec refl_of_cong: (g:ctx) {P:[g ⊢ proc]} [g ⊢ P cong P] =
?
;
```

This declaration introduces a recursive function, denoted as `refl_of_cong`, through the `rec` keyword. This function takes as input any context `g` of schema `ctx` (passed as an implicit argument, due to the round brackets) and any process `P` whose context is `g`; it returns a contextual object of type `[g ⊢ P cong P]`, which represents a derivation of the congruence $P \equiv P$ within the context `g`. The question mark is a computation hole [14] which indicates a subgoal of the proof which remains to be solved; technically, the hole serves as a placeholder for an appropriate computation level object. During the proof search, Beluga can print the typing assumptions of all objects defined thus far, along with the expected type of the hole. This feature aids the user in completing the proof; additionally, interactive mode allows Beluga to automatically fill some of these holes [21].

The next step involves assuming the hypotheses of the theorem.

```
rec refl_of_cong: (g:ctx) {P:[g ⊢ proc]} [g ⊢ P cong P] =
mlam P ⇒ ?
;
```

The `mlam` keyword is used to introduce an explicit universal quantifier; in this example, we introduced a variable `P` of type `[g ⊢ proc]` for some context `g`. Since the context is an implicit argument, it does not need to be introduced through the `mlam` keyword; instead, it is inferred from the declaration of the contextual process `P`.

Next, induction on the structure of the process P is performed through pattern matching on the variable `P`. In Beluga, this is executed through the `case .. of` syntax, with a vertical line introducing each case.

```
rec refl_of_cong: (g:ctx) {P:[g ⊢ proc]} [g ⊢ P cong P] =
mlam P ⇒ case [_ ⊢ P] of
   | [g ⊢ p_zero] ⇒ ?
   | [g ⊢ p_out X Y Q] ⇒ ?
   | [g ⊢ p_in X \y.Q[..,y]] ⇒ ?
   | [g ⊢ Q1 p_par Q2] ⇒ ?
   | [g ⊢ p_res \x.Q[..,x]] ⇒ ?
;
```

In Beluga, the underscore replaces variables which can be reconstructed by the system. In the term `[_ ⊢ P]`, the underscore stands for the context variable, which was not explicitly introduced and therefore does not have a specific name. However, when we introduce each case, we have the freedom to assign names to each variable involved, allowing us to rename the context variable as `g`. We observe that, in the input and restriction cases, meta-functions from `names` to `proc` have been defined through their η-expansion; the body of each function, such as `Q[..,y]` in the input case, is a contextual process whose free variables are included in the weaker context (`g,y:names`). More precisely, the square brackets `[..,y]` denote the substitution of the free variables present in `Q` into the variables of the current context; suspension dots indicate identity substitution. Although explicitly stating context variable substitution is not always necessary (for example, it is not necessary in this proof), we have chosen to include square brackets in all situations in order to always display variable dependencies in a clear way.

Finally, we prove the result for each case.

– In the `p_zero` case, the goal is a contextual object of type `[g ⊢ p_zero cong p_zero]`; this is provided by the object `[g ⊢ c_ref_zero]`.

– In the `p_out` case, we need to exhibit a contextual object of type `[g ⊢ (p_out X Y Q) cong (p_out X Y Q)]`. In the informal proof, the inductive hypothesis is applied to the process Q; this is performed in Beluga through the recursive call of the function `refl_of_cong` on the contextual process `[g ⊢ Q]`. The `let ..=.. in` syntax allows to compute the recursive call and store the outcome in a new contextual object. Hence, by executing the instruction `let [g ⊢ C] = refl_of_cong [g ⊢ Q]` we obtain the result `[g ⊢ C]`, where `C` is a term of type `(Q cong Q)`. Finally, the object `[g ⊢ c_out C]` is a derivation of the desired congruence.

– In the `p_in` case, the goal is to present a contextual object of type `[g ⊢ (p_in X \y.Q[..,y]) cong (p_in X \y.Q[..,y])]`. The informal proof still requires to apply the inductive hypothesis to the process Q; however, in this case, the process `Q` is defined within the context `(g,y:names)`. Thus, we perform the recursive call of the function `refl_of_cong` on the contextual process `[g,y:names ⊢ Q[..,y]]`, obtaining an object `[g,y:names ⊢ C[..,y]]` containing a parametric derivation of the congruence `Q[..,y] cong Q[..,y]`. Finally, the object `[g ⊢ c_in \y.C[..,y]]` is a derivation of the desired congruence.

The `p_par` and `p_res` cases are analogous to the previous ones. We provide the complete proof of the example theorem:

```
rec refl_of_cong: (g:ctx) {P:[g ⊢ proc]} [g ⊢ P cong P] =
/ total p (refl_of_cong g p) /
mlam P ⇒ case [_ ⊢ P] of
  | [g ⊢ p_zero] ⇒ [g ⊢ c_ref_zero]
  | [g ⊢ p_out X Y Q] ⇒ let [g ⊢ C] = refl_of_cong [g ⊢ Q] in
    [g ⊢ c_out C]
  | [g ⊢ p_in X \y.Q[..,y]] ⇒ let [g,y:names ⊢ C[..,y]] =
    refl_of_cong [g,y:names ⊢ Q[..,y]] in [g ⊢ c_in \y.C[..,y]]
  | [g ⊢ Q1 p_par Q2] ⇒ let [g ⊢ C1] = refl_of_cong [g ⊢ Q1] in
    [g ⊢ c_par C1]
  | [g ⊢ p_res \x.Q[..,x]] ⇒ let [g,x:names ⊢ C[..,x]] =
    refl_of_cong [g,x:names ⊢ Q[..,x]] in [g ⊢ c_res \x.C[..,x]]
;
```

Figure 3.6: Proof of the example theorem.

The second line specifies that the `refl_of_cong` function takes two arguments `g` and `p` (where the latter represents the process `P:[g ⊢ proc]`), it is decreasing on the second argument `p` and it is total, meaning it is defined on all inputs and always terminates. In order to streamline the notation, underscores can replace arguments which are not decreasing. This annotation enables the Beluga totality checker to confirm that each case has been covered and that each recursive call is structurally decreasing on the specified argument. Such line is necessary to ensure that a recursive function represents a proof.

3.3.1 Theorem 1: τ-Transition Implies Reduction

Before presenting the formalization of the preliminary lemmas for Theorem 1, we need to make a remark. We recall that the Beluga formalization implements the late LTS semantics, while the informal description of the lemmas in the benchmark follows the early approach. Since Lemma 1.1 concerns input transitions, which are interpreted differently depending on the semantics approach, its statement requires a slight adjustment. We present its modified statement for the late semantics:

Lemma 1.1'. ***Rewriting of processes involved in input transitions***
If $Q \xrightarrow{x(y)} Q'$ then there exist a finite (possibly empty) set of names $w_1, ..., w_n$ (with $x, y \neq w_i\ \forall i = 1, ..., n$) and two processes R, S such that $Q \equiv (\nu w_1)...(\nu w_n)(x(y).R \mid S)$ and $Q' \equiv (\nu w_1)...(\nu w_n)(R \mid S)$.

The conclusion of this lemma includes an existential quantification; however, Beluga lacks built-in keywords or syntax to formally express this construct. A

typical solution to this problem involves defining a new type family which encodes the existential quantification. This allows proving the lemma by constructing a derivation of an object of such type.
Another issue consists in encoding telescopes [3], i.e. the sequences of an indefinite amount of binders $(\nu w_1)...(\nu w_n)$ restricting the parallel compositions $(x(y).R \mid S)$ and $(R \mid S)$ occurring in the two processes congruent to Q and Q'. This issue is resolved by treating telescopes inductively. In the base case, Q and Q' are congruent to processes without restrictions; in the inductive case, they are congruent to single restrictions $(\nu w)P$ and $(\nu w)P'$, where P and P' inductively have the desired form.

We define the type family ex_inp_rew which encodes the existence of names and processes as in the statement of Lemma 1.1':

```
LF ex_inp_rew: proc → names → (names → proc) → type =
  | inp_base: Q cong ((p_in X R) p_par S)
    → ({y:names} (Q' y) cong ((R y) p_par S)) → ex_inp_rew Q X Q'
  | inp_ind: Q cong (p_res P) → ({y:names} (Q' y) cong (p_res (P' y)))
    → ({w:names} ex_inp_rew (P w) X \y.(P' y w)) → ex_inp_rew Q X Q'
;
```

The type ex_inp_rew is indexed by a process, a name, and a process abstraction (i.e. a function from names to processes). These three arguments represent the elements which compose an input transition (recall that the result of a bound transition is actually a process abstraction). The first constructor inp_base estabilishes the following fact: if Q cong ((p_in X R) p_par S) holds for some process S and process abstraction R, and if (Q' y) cong ((R y) p_par S) holds for every name y, then ex_inp_rew Q X Q' holds. We observe that the universal quantification in the second premise is necessary. From a syntactical perspective, indeed, both Q' and R are process abstractions; therefore, they must be applied to some name in order to obtain processes that can occur in a congruence. From a semantical perspective, Q' and R represent processes with a placeholder for some input name to be further received; thus, the congruence involving these two processes must hold for any input name substitution. The second constructor inp_ind states that, if Q and Q' are congruent to restrictions $(\nu w)P$ and $(\nu w)P'$ respectively and ex_inp_rew P X P' (with a minor abuse of notation) holds for any choice of binder name w, then ex_inp_rew Q X Q' holds.

Lemma 1.1': bs_in_rew

We present the proof of Lemma 1.1' in Figure 3.7.

Proof Verbalization: The recursive function bs_in_rew takes as arguments any context g of schema ctx, any process Q in the context g and an object of type [g ⊢ bstep Q (b_in X) \y.Q'[..,y]]; consequently, the function implicitly takes the arguments X and Q' as well. It returns an object of type [g ⊢ ex_inp_rew Q X \y.Q'[..,y]] and it is decreasing on the second argument Q.
We introduce a variable Q, representing the universally quantified process, and a variable b denoting the explicit hypothesis [g ⊢ bstep Q (b_in X) \y.Q'[..,y]]. As the latter is not explicitly universally quantified, the variable b is introduced using the fn keyword. Next, we implement case analysis on Q through pattern matching, obtaining five cases.

```
rec bs_in_rew: (g:ctx) {Q:[g ⊢ proc]} [g ⊢ bstep Q (b_in X) \y.Q'[..,y]]
   → [g ⊢ ex_inp_rew Q X \y.Q'[..,y]] =
/ total q (bs_in_rew _ q _ _ _) /
mlam Q ⇒ fn b ⇒ case [_ ⊢ Q] of
   | [g ⊢ p_zero] ⇒ impossible b
   | [g ⊢ p_in W \z.P[..,z]] ⇒ let [g ⊢ bs_in] = b in
     [g ⊢ inp_base (c_sym par_unit) \y.(c_sym par_unit)]
   | [g ⊢ p_out W Z P] ⇒ impossible b
   | [g ⊢ Q1 p_par Q2] ⇒
     (case b of
        | [g ⊢ bs_par1 B1] ⇒
          let [g ⊢ D1] = bs_in_rew [g ⊢ Q1] [g ⊢ B1] in
          let [g ⊢ D2] = bs_in_rew_par1 [g ⊢ Q2] [g ⊢ D1] in [g ⊢ D2]
        | [g ⊢ bs_par2 B2] ⇒
          let [g ⊢ D1] = bs_in_rew [g ⊢ Q2] [g ⊢ B2] in
          let [g ⊢ D2] = bs_in_rew_par2 [g ⊢ Q1] [g ⊢ D1] in [g ⊢ D2]
      )
   | [g ⊢ p_res \z.P[..,z]] ⇒ let [g ⊢ bs_res \z.B[..,z]] = b in
     let [g,z:names ⊢ D1[..,z]] = bs_in_rew [g,z:names ⊢ P[..,z]]
     [g,z:names ⊢ B[..,z]] in
     let [g ⊢ D2] = bs_in_rew_res [g,z:names ⊢ D1[..,z]] in [g ⊢ D2]
;
```

Figure 3.7: Proof of Lemma 1.1'.

– The `[g ⊢ p_zero]` and `[g ⊢ p_out W Z P]` cases would imply that `b` is an object of type `[g ⊢ bstep p_zero (b_in X) \y.Q'[..,y]]` or `[g ⊢ bstep (p_out W Z P) (b_in X) \y.Q'[..,y]]`; however, these types are not inhabited, leading to an absurd. The `impossible` keyword, followed by `b`, is used to perform pattern matching on `b` when no constructor matches the structure of `b`; since no case has to be addressed, the conclusion is reached.

– In the `[g ⊢ p_in W \z.P[..,z]]` case, `b` is an object of type `[g ⊢ bstep (p_in W \z.P[..,z]) (b_in X) \y.Q'[..,y]]`. We perform inversion on `b` using the `let` keyword, specifying that this object must have been built through the `bs_in` constructor. Through this operation, Beluga infers that the type of `b` is `[g ⊢ bstep (p_in X \z.P[..,z]) (b_in X) \z.P[..,z]]` and performs unification of all the involved variables: for example, it deduces that `Q'` equals to `P` and that the goal is `[g ⊢ ex_inp_rew (p_in X \z.P[..,z]) X \z.P[..,z]]`. We note that `c_sym par_unit`, when applied to the appropriate implicit arguments, is an object of type `(p_in X \z.P[..,z]) cong ((p_in X \z.P[..,z]) p_par p_zero)`; analogously, `c_sym par_unit` denotes the congruence of the process `P[..,y]` to `P[..,y] p_par p_zero` for any `y`. As a result, the object `[g ⊢ inp_base (c_sym par_unit) \y.(c_sym par_unit)]` has the desired type.

– In the `[g ⊢ Q1 p_par Q2]` case, `b` is an object of type `[g ⊢ bstep (Q1 p_par Q2) (b_in X) (\y.Q'[..,y])]`. By inversion on the variable `b`, we obtain two subcases `[g ⊢ bs_par1 B1]` and `[g ⊢ bs_par2 B2]`. In each subcase, either the left component Q_1 performs an input transition $Q_1 \xrightarrow{x(y)} R_1$, encoded by the variable B1, or the right component Q_2 performs an input transition encoded by B2. We describe the first case only, as the other case is analogous. We apply a recursive call of the bs_in_rew function on the arguments [g ⊢ Q1] and [g ⊢ B1], obtaining an object [g ⊢ D1] of type [g ⊢ ex_inp_rew Q1 X \y.R1[..,y]] which expresses the rewriting of the processes involved in the transition $Q_1 \xrightarrow{x(y)} R_1$. To conclude, we aim to obtain an object of type ex_inp_rew encoding the rewriting for the parallel composition $Q_1 \mid Q_2$. However, to achieve this, we need to unfold the object [g ⊢ D1] and handle explicit congruences. For this reason, we delegate this task to an auxiliary lemma bs_in_rew_par1 (respectively, bs_in_rew_par2 in the bs_par2 case). Once this lemma is stated and proved, we reach the conclusion.

– In the [g ⊢ p_res \z.P[..,z]] case, the proof follows a similar pattern to the previous case. Initially, we perform inversion on b through the bs_res constructor, obtaining a variable B representing the input transition of the subprocess P. Next, we apply the recursive call of the bs_in_rew function on this transition, operating in a weaker context (g,z:names), and obtaining an object of type ex_inp_rew which represents the corresponding rewriting. Finally, we use the auxiliary lemma bs_in_rew_res to unfold this object and conclude the proof.

□

We now present the proof of the first auxiliary lemma, bs_in_rew_par1. In mathematical terms, it corresponds to the following result: "given two processes Q and Q' such that $Q \equiv (\nu\overline{w})(x(y).P \mid S)$ and $Q' \equiv (\nu\overline{w})(P \mid S)$ for some P, S and $\overline{w}$, then it can be proved that, for any process R in which each w_i does not occur free, $Q \mid R \equiv (\nu\overline{w})(x(y).P' \mid S')$ and $Q' \mid R \equiv (\nu\overline{w})(P' \mid S')$ for some processes P', S'". In our case, we are going to show that $P' := P$ and $S' := S \mid R$.

```
rec bs_in_rew_par1:(g:ctx){R:[g ⊢ proc]}[g ⊢ ex_inp_rew Q X \y.Q'[..,y]]
    → [g ⊢ ex_inp_rew (Q p_par R) X \y.(Q'[..,y] p_par R[..])] =
/ total d (bs_in_rew_par1 _ _ _ _ _ _ d) /
mlam R ⇒ fn d ⇒ case d of
   | [g ⊢ inp_base C1 \y.C2[..,y]] ⇒ [g ⊢ inp_base (c_trans (c_par C1)
     (c_sym par_assoc)) \y.(c_trans (c_par C2[..,y]) (c_sym par_assoc))]
   | [g ⊢ inp_ind C1 (\y.C2[..,y]) (\w.D1[..,w])] ⇒
     let [g,w:names ⊢ D2[..,w]] = bs_in_rew_par1 [g,w:names ⊢ R[..]]
     [g,w:names ⊢ D1[..,w]] in [g ⊢ inp_ind (c_trans (c_par C1)
     sc_ext_par) (\y.(c_trans (c_par C2[..,y]) sc_ext_par)) (\w.D2[..,w])]
;
```

Proof Verbalization: Given an object of type [g ⊢ ex_inp_rew Q X \y.Q'[..,y]], the goal consists in obtaining an object of type [g ⊢ ex_inp_rew (Q p_par R) X \y.(Q'[..,y] p_par R[..])] for any process R. The process R must be explicitly passed as an argument to the bs_in_rew_par1 function: otherwise, Beluga would not be able to reconstruct it during recursive calls. After introducing the hypotheses R and d, we unfold the object d by pattern matching.

– In the `inp_base` case, the additional hypotheses `C1` and `C2` denote objects of type `[g ⊢ Q cong ((p_in X S) p_par T)]` and `[g,y:names ⊢ (Q' y) cong ((S y) p_par T)]`. To construct an `ex_inp_rew` object for $Q \mid R$ and $Q' \mid R$, we build the appropriate congruences for these processes as outlined in the informal proof on page 14, and then pass them to the `inp_base` constructor.

– In the `inp_ind` case, `C1` and `C2` denote objects of type `[g ⊢ Q cong (p_res P)]` and `[g,y:names ⊢ (Q' y) cong (p_res (P' y))]`, while `D1` is a derivation of `[g,w:names ⊢ ex_inp_rew (P w) X \y.(P' y w)]`. We recursively apply the function `bs_in_rew_par1` to the arguments `R` and `D1` in the weaker context `(g,w:names)`, obtaining a parametric derivation `D2` of `[g,w:names ⊢ ex_inp_rew ((P w) p_par R[..]) X \y.((P' y w) p_par R[..])]`; we observe that, when passing the object `[g,w:names ⊢ R[..]]` in the recursive call, the square brackets are necessary to weaken the process `R`, originally defined in the stronger context `g`, to use it in the context `(g,w:names)`. Finally, we conclude by passing the appropriate congruences and the object `D2` to the `inp_ind` constructor.

□

We provide the statement of the other two auxiliary lemmas, `bs_in_rew_par2` and `bs_in_rew_res`. Given their similarity to the previous lemma, we omit their proof and description.

```
rec bs_in_rew_par2:(g:ctx){R:[g ⊢ proc]}[g ⊢ ex_inp_rew Q X \y.Q'[..,y]]
    → [g ⊢ ex_inp_rew (R p_par Q) X \y.(R[..] p_par Q'[..,y])] = ?
;

rec bs_in_rew_res: (g:ctx) [g,z:names ⊢ ex_inp_rew Q[..,z] X[..]
    \y.Q'[..,z,y]] → [g ⊢ ex_inp_rew (p_res \z.Q[..,z]) X \y.(p_res
    \z.Q'[..,z,y])] = ?
;
```

Lemma 1.2 and 1.3: `fs_out_rew` and `bs_out_rew`

The encoding of the other two preliminary lemmas, Lemma 1.2 and Lemma 1.3, follows the same pattern: first, defining a new type family for the rewriting of processes involved in free or bound output transitions, with base case and inductive case constructors to encode the telescopes; next, constructing the main recursive function which proves the lemma by induction on the structure of processes; finally, stating and proving auxiliary lemmas in order to prove the result in specific subcases. It is worth noting that, analogously to the observation on page 15 for the informal proof, the `bs_open` case for the `bs_out_rew` proof requires calling the recursive function `fs_out_rew` and defining an additional auxiliary lemma. We present the definition of the type families `ex_fout_rew` and `ex_bout_rew`, as well as the declaration of the recursive functions `fs_out_rew` and `bs_out_rew`:

```
LF ex_fout_rew: proc → names → names → proc → type =
  | fout_base: Q cong ((p_out X Y R) p_par S) → Q' cong (R p_par S)
    → ex_fout_rew Q X Y Q'
  | fout_ind: Q cong (p_res P) → Q' cong (p_res P')
    → ({w:names} ex_fout_rew (P w) X Y (P' w)) → ex_fout_rew Q X Y Q'
;
```

```
LF ex_bout_rew: proc → names → (names → proc) → type =
  | bout_base: Q cong (p_res \z.((p_out X z (R z)) p_par (S z)))
    → ({y:names} (Q' y) cong ((R y) p_par (S y))) → ex_bout_rew Q X Q'
  | bout_ind: Q cong (p_res P) → ({y:names} (Q' y) cong (p_res (P' y)))
    → ({w:names} ex_bout_rew (P w) X \y.(P' y w)) → ex_bout_rew Q X Q'
;

rec fs_out_rew: (g:ctx) {Q:[g ⊢ proc]} [g ⊢ fstep Q (f_out X Y) Q']
   → [g ⊢ ex_fout_rew Q X Y Q'] = ?
;

rec bs_out_rew:(g:ctx){Q:[g ⊢ proc]} [g ⊢ bstep Q (b_out X) \y.Q'[..,y]]
   → [g ⊢ ex_bout_rew Q X \y.Q'[..,y]] = ?
;
```

Theorem 1: `fstep_impl_red`

We provide the encoding of the proof of Theorem 1, which states that $P \xrightarrow{\tau} Q$ implies $P \to Q$:

```
rec fstep_impl_red: (g:ctx) [g ⊢ fstep P f_tau Q] → [g ⊢ P red Q] =
/ total f (fstep_impl_red _ _ _ f) /
fn f ⇒ case f of
   | [g ⊢ fs_par1 F1] ⇒ let [g ⊢ R] = fstep_impl_red [g ⊢ F1] in
                         [g ⊢ r_par R]
   | [g ⊢ fs_par2 F2] ⇒ let [g ⊢ R] = fstep_impl_red [g ⊢ F2] in
                         [g ⊢ r_str par_comm (r_par R) par_comm]
   | [g ⊢ fs_com1 F1 B1] ⇒
     let [g ⊢ D1] = fs_out_rew [g ⊢ _] [g ⊢ F1] in
     let [g ⊢ D2] = bs_in_rew [g ⊢ _] [g ⊢ B1] in
     let [g ⊢ R] = fs_com1_impl_red [g ⊢ D1] [g ⊢ D2] in [g ⊢ R]
   | [g ⊢ fs_com2 B1 F1] ⇒
     let [g ⊢ D1] = bs_in_rew [g ⊢ _] [g ⊢ B1] in
     let [g ⊢ D2] = fs_out_rew [g ⊢ _] [g ⊢ F1] in
     let [g ⊢ R] = fs_com2_impl_red [g ⊢ D1] [g ⊢ D2] in [g ⊢ R]
   | [g ⊢ fs_res \z.F[..,z]] ⇒
     let [g,z:names ⊢ R[..,z]] = fstep_impl_red [g,z:names ⊢ F[..,z]] in
     [g ⊢ r_res \z.R[..,z]]
   | [g ⊢ fs_close1 B1 B2] ⇒
     let [g ⊢ D1] = bs_out_rew [g ⊢ _][g ⊢ B1] in
     let [g ⊢ D2] = bs_in_rew [g ⊢ _] [g ⊢ B2] in
     let [g ⊢ R] = fs_close1_impl_red [g ⊢ D1] [g ⊢ D2] in [g ⊢ R]
   | [g ⊢ fs_close2 B1 B2] ⇒
     let [g ⊢ D1] = bs_in_rew [g ⊢ _] [g ⊢ B1] in
     let [g ⊢ D2] = bs_out_rew [g ⊢ _] [g ⊢ B2] in
     let [g ⊢ R] = fs_close2_impl_red [g ⊢ D1] [g ⊢ D2] in [g ⊢ R]
;
```

Proof Verbalization: The function `fstep_impl_red` takes a term of type `[g ⊢ fstep P f_tau Q]` as an argument and returns an object of type `[g ⊢ P red Q]`. It is decreasing on the former argument. The proof proceeds by introducing the variable `f`, which denotes the τ-transition, and performing pattern matching on it. In the two `fs_par` and in the `fs_res` cases, it is sufficient to recursively apply the `fstep_impl_red` function on a structurally smaller transition and conclude using the appropriate `red` constructor (`r_par`, `r_str` and `r_res` respectively). In the `fs_com` and `fs_close` cases, we have some input and output transitions as hypotheses: hence we apply the previously defined functions `bs_in_rew`, `fs_out_rew` and `bs_out_rew` to obtain two objects, `D1` and `D2`, which contain information about the structure of the processes involved in the two transitions. Finally, we need to unfold `D1` and `D2`, in order to show that these processes can perform a reduction: this is achieved by calling an additional lemma for each case. □

We introduce the encoding of the lemma `fs_com1_impl_red` applied in the `fs_com1` case; the other lemmas are formalized in a similar way. Prior to presenting the code, we first discuss the proof technique employed.

The `fs_com1_impl_red` function takes two objects of type `[g ⊢ ex_fout_rew P1 X Y Q1]` and `[g ⊢ ex_inp_rew P2 X \x.Q2[..,x]]` as arguments, and returns an object of type `[g ⊢ (P1 p_par P2) red (Q1 p_par Q2[..,Y])]`. In mathematical terms it states that, given processes P_1, P_2, Q_1 and Q_2 which satisfy the congruences on page 16, then $P_1 \mid P_2 \to Q_1 \mid Q_2$. The proof requires to perform inversion on both hypotheses, addressing both the base case and inductive case for each of the types `ex_inp_rew` and `ex_fout_rew`. When considering the inductive case for either type, the `fs_com1_impl_red` function needs to be recursively called on a structurally smaller object of that type. This implies that the `fs_com1_impl_red` function must be decreasing on both arguments; according to the Curry-Howard isomorphism, this corresponds to employing double induction. However, Beluga lacks support for functions decreasing on more than one argument, thus precluding this method.

In order to clarify the solution adopted for the encoding of double recursion, we first present the description of its mathematical counterpart. The following theorem outlines one possible approach to prove that a property P, depending on a pair of natural numbers n and m, holds true for any n and m:

Theorem.
Let $P(n, m)$ be some property, indexed by $n, m \in \mathbb{N}$, such that:

1. *$P(0, m)$ is true $\forall m \in \mathbb{N}$;*
2. *If $P(n, m)$ is true for some $n, m \in \mathbb{N}$, then $P(n + 1, m)$ is also true;*

Then $P(n, m)$ is true $\forall n, m \in \mathbb{N}$.

Although distinct from double induction or lexicographic induction, the theorem is equivalent to these two principles. We note that the first premise of this theorem can be achieved through single induction on the second parameter when the first parameter equals 0. Conversely, the second premise represents the inductive step for induction on the first parameter, for each $m \in \mathbb{N}$ as a second parameter.
This principle suggests that double recursion can be implemented by decomposing it into the definition of two single recursive functions. The first function handles the base case of the first input type and is decreasing on the structure of the second type; the second function is decreasing on the structure of the first type, and relies on the call of the first recursive function to address the base case.

We present the encoding of the `fs_com1_impl_red` function, decreasing on the first argument, and the encoding of the `fs_com1_impl_red_base` function, which handles the base case of the first argument and is decreasing on the second argument:

```
rec fs_com1_impl_red_base: (g:ctx) [g ⊢ P2 cong ((p_in X \x.R[..,x])
  p_par S)] → [g,w:names ⊢ Q2[..,w] cong (R[..,w] p_par S[..])]
  → [g ⊢ ex_fout_rew P1 X Y Q1]
  → [g ⊢ (P1 p_par P2) red (Q1 p_par Q2[..,Y])] =
/ total d1 (fs_com1_impl_red_base _ _ _ _ _ _ _ _ _ _ _ _ d1) /
fn c3 ⇒ fn c4 ⇒ fn d1 ⇒ case d1 of
   | [g ⊢ fout_base C1 C2] ⇒
     let [g ⊢ C3] = c3 in
     let [g,w:names ⊢ C4[..,w]] = c4 in
     [g ⊢ r_str (c_trans (c_par C1) (c_trans par_comm (c_trans (c_par C3)
     par_comm))) (r_str par_assoc (r_par (r_str (c_trans (c_par par_comm)
     (c_trans (c_sym par_assoc) par_comm)) (r_par r_com) (c_trans par_comm
     (c_trans par_assoc (c_par par_comm))))) (c_sym par_assoc)) (c_trans
     (c_par (c_sym C2)) (c_trans par_comm (c_trans (c_par
     (c_sym C4[..,_])) par_comm)))]
   | [g ⊢ fout_ind C1 C2 \w.D1[..,w]] ⇒
     let [g ⊢ C3] = c3 in
     let [g,y:names ⊢ C4[..,y]] = c4 in
     let [g,w:names ⊢ R1[..,w]] = fs_com1_impl_red_base
     [g,w:names ⊢ C3[..]] [g,w:names,y:names ⊢ C4[..,y]]
     [g,w:names ⊢ D1[..,w]] in
     [g ⊢ r_str (c_trans (c_par C1) sc_ext_par) (r_res \w.R1[..,w])
     (c_trans (c_sym sc_ext_par) (c_par (c_sym C2)))]
;
```

Note that the base case of this lemma, corresponding to the base case of both types `ex_inp_rew` and `ex_fout_rew`, is addressed through a complex chain of congruences and reductions, mirroring the structure of the informal proof provided on page 16.

```
rec fs_com1_impl_red: (g:ctx) [g ⊢ ex_fout_rew P1 X Y Q1]
  → [g ⊢ ex_inp_rew P2 X \x.Q2[..,x]]
  → [g ⊢ (P1 p_par P2) red (Q1 p_par Q2[..,Y])] =
/ total d2 (fs_com1_impl_red _ _ _ _ _ _ _ _ _ d2) /
fn d1 ⇒ fn d2 ⇒ case d2 of
   | [g ⊢ inp_base C3 \y.C4[..,y]] ⇒
     let [g ⊢ R] = fs_com1_impl_red_base [g ⊢ C3]
     [g,y:names ⊢ C4[..,y]] d1 in [g ⊢ R]
   | [g ⊢ inp_ind C3 (\y.C4[..,y]) (\w.D2[..,w])] ⇒
     let [g ⊢ D1] = d1 in
     let [g,w:names ⊢ R1[..,w]] = fs_com1_impl_red [g,w:names ⊢ D1[..]]
     [g,w:names ⊢ D2[..,w]] in
     [g ⊢ r_str (c_trans par_comm (c_trans (c_par C3) (c_trans sc_ext_par
     (c_res \w.par_comm)))) (r_res \w.R1[..,w]) (c_trans (c_res
     \w.par_comm) (c_trans (c_sym sc_ext_par) (c_trans (c_par
     (c_sym C4[..,_])) par_comm)))]
;
```

3.3.2 Theorem 2: Reduction Implies τ-Transition

In the proof of Theorem 2, we rely on two main lemmas: Lemma 2.6 and Lemma 2.7. Lemma 2.6 requires five auxiliary lemmas regarding free and bound names in processes and transitions. The majority of these auxiliary lemmas does not necessitate explicit formulation and proof in Beluga. In fact, these lemmas were applied in the proof of Lemma 2.6 with the purpose of verifying the satisfaction of side conditions for transition rules. However, as previously explained in section 3.2.2, explicit enunciation of side conditions in the Beluga encoding is not necessary, due to the employment of higher-order abstract syntax; as a result, during a proof, the system automatically determines whether these side conditions are met or not.

While Lemmas 2.1, 2.2, 2.3 and 2.5 do not require an explicit statement, the Beluga encoding relies on the following result, related to Lemma 2.4: "given a contextual object of type `[g,x:names ⊢ fstep P[..] A Q]`, then it can be shown that `[g ⊢ fstep P A' Q']` holds, with `A'[..]=A` and `Q'[..]=Q` in the context `(g,x:names)`. The same holds true for bound transitions as well". In other words, this statement asserts that, given a transition $P \xrightarrow{\alpha} Q$ where x does not occur free in P nor bound in any other entity (due to variable conventions), then x does not occur free in either α and Q. Since this result involves strengthening the context in which an object is defined, it can be referred to as a "strengthening lemma".

Strengthening Lemma: `strengthen_fstep` and `strengthen_bstep`

We begin with the definition of three type families encoding equality of processes, free actions and bound actions. Since the Logical Frameworks LF do not provide any built-in equality notion, we must define a type family representing equality for each of these types. Each type family presents only one constructor, which relates equal terms. We only present the definition of the type family `eqp` which encodes equality of processes; the other type families `eqf` and `eqb` for free and bound actions are defined analogously.

```
LF eqp: proc → proc → type =
  | prefl: eqp P P
;
```

Next, we need to define two type families encoding the existence of a transition in a stronger context, given an initial transition in a weaker context with the hypotheses described above. However, unlike the existential definitions in the previous section, we are now dealing with a property about a LF contextual object (the initial transition); this implies that the new type families must be parametrized by a contextual object. Although this cannot be done at the LF level, this judgement can be encoded at the computation level [21].
The corresponding type families `ex_str_fstep` and `ex_str_bstep` are defined through the following declaration:

```
inductive ex_str_fstep: (g:ctx) [g,x:names ⊢ fstep P[..] A Q] → ctype =
  | ex_fstep: {F:[g,x:names ⊢ fstep P[..] A Q]} [g ⊢ fstep P A' Q']
    → [g,x:names ⊢ eqf A A'[..]] → [g,x:names ⊢ eqp Q Q'[..]]
    → ex_str_fstep [g,x:names ⊢ F]
;
```

```
inductive ex_str_bstep: (g:ctx) [g,x:names ⊢ bstep P[..] A \z.Q[..,x,z]]
    → ctype =
  | ex_bstep: {B:[g,x:names ⊢ bstep P[..] A \z.Q[..,x,z]]}
    [g ⊢ bstep P A' \z.Q'[..,z]] → [g,x:names ⊢ eqb A A'[..]]
    → [g,x:names,z:names ⊢ eqp Q[..,x,z] Q'[..,z]]
    → ex_str_bstep [g,x:names ⊢ B]
```

The `inductive .. ctype` syntax is used to define inductive type families at the computation level in Beluga. The type family `ex_str_fstep` is indexed by an object of type `[g,x:names ⊢ fstep P[..] A Q]`, representing a free transition in the context `(g,x:names)` where `x` does not occur free in `P`; additionally, the type family is implicitly parametrized by the context `g` and by the other entities appearing in the transition. The constructor `ex_fstep` estabilishes that, for any `F` of type `[g,x:names ⊢ fstep P[..] A Q]`, we have that `ex_str_fstep F` holds true in case `[g ⊢ fstep P A' Q']` is verified for some objects `A'` and `Q'`; these objects, defined in the context `g`, must be respectively equal to `A` and `Q` in the weaker context `(g,x:names)`. The definition of the `ex_str_bstep` type family for the existence of bound transitions may appear more complex due to the increased amount of variable dependencies, but its meaning is analogous to the previous definition.

The strengthening lemma is encoded through the definition of two mutually recursive functions, `strengthen_fstep` and `strengthen_bstep`, which take an object representing a free or bound transition and return an appropriate object of type `ex_str_fstep` or `ex_str_bstep`, respectively. In this section, we present only the declaration of these functions; the proof term is left to Appendix C.

```
rec strengthen_fstep: (g:ctx) {F:[g,x:names ⊢ fstep P[..] A Q]}
  → ex_str_fstep [g,x:names ⊢ F] = ?

and rec strengthen_bstep: (g:ctx) {B:[g,x:names ⊢ bstep P[..] A
  \z.Q[..,x,z]]} → ex_str_bstep [g,x:names ⊢ B] = ?
;
```

Lemma 2.6: `cong_fstepleft_impl_fstepright` (and others)

We recall the statement of Lemma 2.6: "if $P \equiv Q$ and $P \xrightarrow{\alpha} P'$, then there exists a process Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \equiv Q'$". It was proved along with its symmetrical counterpart: "if $P \equiv Q$ and $Q \xrightarrow{\alpha} Q'$, then there exists a process P' such that $P \xrightarrow{\alpha} P'$ and $P' \equiv Q'$". The following diagrams display the two statements:

$$\begin{tikzcd}
P \arrow[r, phantom, "\equiv"] \arrow[d, "\alpha"] & Q \arrow[d, dashed, "\alpha"] \\
P' \arrow[r, phantom, "\equiv"] & Q'
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
P \arrow[r, phantom, "\equiv"] \arrow[d, dashed, "\alpha"] & Q \arrow[d, "\alpha"] \\
P' \arrow[r, phantom, "\equiv"] & Q'
\end{tikzcd}$$

As the conclusion of the lemma includes an existential quantification, we need to define a new type family encoding it. However, since the statement involves transitions through a generic action α which can be either free or bound, we actually require two new type families: one for free transitions and one for bound transitions.

```
LF ex_fstepcong: proc → proc → f_act → proc → type =
  | fsc: fstep Q A Q’ → P’ cong Q’ → ex_fstepcong P Q A P’
;

LF ex_bstepcong: proc → proc → b_act → (names → proc) → type =
  | bsc: bstep Q A Q’ → ({x:names} (P’ x) cong (Q’ x))
  → ex_bstepcong P Q A P’
;
```

The constructor `fsc` estabilishes that `ex_fstepcong P Q A P’` holds if there exists a process `Q’` such that `fstep Q A Q’` and `P’ cong Q’` hold. Similarly, the constructor `bsc` estabilishes that `ex_bstepcong P Q A P’` holds if there exists a process abstraction `Q’` such that `bstep Q A Q’` holds and `(P’ x) cong (Q’ x)` holds for any name `x`. We observe that the process `P` does not play any role in the definitions of the `fsc` and `bsc` constructors: for this reason, the two type families could be defined without the first argument `P`. Nevertheless, we are going to keep the redundant definition of the two type families, as the more concise definition leads to a coverage error in a further demonstration. The cause of the coverage error, detected by the Beluga totality checker, remains unknown.

Lemma 2.6 is encoded through the definition of the following four mutual recursive functions: `cong_fstepleft_impl_stepright`, `cong_fstepright_impl_stepleft`, `cong_bstepleft_impl_stepright` and `cong_bstepright_impl_stepleft`. Since the lemma statement involves transitions via a generic action α, the first two recursive functions prove the result for free transitions, while the last two functions demonstrate the result for bound transitions. Moreover, since the proof is carried out by concurrently estabilishing two symmetrical assertions, the odd functions prove the “left” statement, while the even functions demonstrate the “right” statement. We present the declaration of these four recursive functions, along with the proof terms corresponding to the same cases discussed in the informal proof on page 21; totality annotations are omitted, as we are reporting a partial proof. The complete demonstration is provided in Appendix C.

We begin with the first function, `cong_fstepleft_impl_stepright`, which proves the “left” assertion for free transitions. The corresponding proof is provided in Figure 3.8.

Proof Verbalization: The proof proceeds by induction on the structure of the congruence `c`.

– In the `par_assoc` case, `c` represents the congruence $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$. We begin by pattern matching on the free transition `f`, which is a transition starting from the process $P \mid (Q \mid R)$; here, we only address the case in which `f` has the form `fs_par2 F`, for some free transition `F` starting from the process $Q \mid R$. We proceed with another pattern matching on `F`, considering only the case where it has the form `fs_close1 B1 B2`; `B1` denotes a bound output transition from Q and `B2` denotes a bound input transition from R. At this stage, we conclude by exhibiting the encodings of the required transition and congruence, provided in the informal proof on page 22; these two objects are embedded within the `fsc` constructor, in order to return an object of type `ex_fstepcong`.

```
rec cong_fstepleft_impl_fstepright: (g:ctx) [g ⊢ P cong Q]
  → [g ⊢ fstep P A P'] → [g ⊢ ex_fstepcong P Q A P'] =
fn c ⇒ fn f ⇒ case c of
   | [g ⊢ par_assoc] ⇒
     (case f of
        | [g ⊢ fs_par2 F] ⇒
          (case [g ⊢ F] of
             | [g ⊢ fs_close1 B1 B2] ⇒
               [g ⊢ fsc (fs_close1 (bs_par2 B1) B2) (c_trans
               par_comm (c_trans sc_ext_par (c_trans (c_res
               \x.par_comm) (c_res \x.par_assoc))))]
          )
      )
   | [g ⊢ sc_ext_par] ⇒
     (case f of
        | [g ⊢ fs_par2 F2] ⇒
          [g ⊢ fsc (fs_res \x.(fs_par2 F2[..])) sc_ext_par]
     )
   | [g ⊢ c_par C1] ⇒
     (case f of
        | [g ⊢ fs_par2 F1] ⇒ [g ⊢ fsc (fs_par2 F1) (c_par C1)]
     )
   | [g ⊢ c_sym C'] ⇒
     let [g ⊢ D] = cong_fstepright_impl_fstepleft [g ⊢ C'] f in [g ⊢ D]
```

Figure 3.8: Proof of Lemma 2.6 (first part).

- The proof of the `sc_ext_par` and `c_par` subcases presented here is quite straightforward: after performing inversion on `c` and `f`, we can immediately provide the required transition and congruence. Unlike the informal proof, there is no need to apply additional lemmas regarding free and bound variables, since variable dependencies are automatically inferred by the system.

- The `c_sym` case, which was originally not selected in the informal proof on Chapter 2, is presented here in order to exhibit the recursive call of the symmetrical function `cong_fstepright_impl_stepleft`, which is sufficient to conclude.

□

We provide the proof of the lemma `cong_fstepright_impl_stepleft`, which proves the "right" assertion for free transitions, in Figure 3.9.

Proof Verbalization: The proof proceeds by induction on the structure of the congruence `c`.

- In the `sc_ext_par` case, `c` represents the congruence $(\nu x)P \mid Q \equiv (\nu x)(P \mid Q)$ where x does not occur free in Q. By inversion on `f`, which denotes a transition starting from $(\nu x)(P \mid Q)$, we obtain that it can only have the structure `[g ⊢ fs_res \x.F[..,x]]`: the S-Open case present in the informal proof is handled by the `cong_bstepright_impl_stepleft` function, as it involves a

```
and rec cong_fstepright_impl_fstepleft: (g:ctx) [g ⊢ P cong Q]
  → [g ⊢ fstep Q A Q'] → [g ⊢ ex_fstepcong Q P A Q'] =
fn c ⇒ fn f ⇒ case c of
   | [g ⊢ sc_ext_par] ⇒ let [g ⊢ fs_res \x.F[..,x]] = f in
     (case [g,x:names ⊢ F[..,x]] of
        | [g,x:names ⊢ fs_com1 F1[..,x] B1[..,x]] ⇒
          let ex_bstep [g,x:names ⊢ B1[..,x]] [g ⊢ B1'] e1 e2 =
          strengthen_bstep [g,x:names ⊢ B1[..,x]] in
          let [g,x:names ⊢ brefl] = e1 in
          let [g,x:names,z:names ⊢ prefl] = e2 in
          let [g,x:names ⊢ F1[..,x]]:[g,x:names ⊢ fstep P (f_out
          Z[..] W) P'] = [g,x:names ⊢ F1[..,x]] in
          (case [g,x:names ⊢ W] of
             | [g,x:names ⊢ #w[..]] ⇒ [g ⊢ fsc (fs_com1 (fs_res
               \x.F1[..,x]) B1') (c_sym sc_ext_par)]
             | [g,x:names ⊢ x] ⇒ [g ⊢ fsc (fs_close1 (bs_open
               \x.F1[..,x]) B1') c_ref]
          )
        | [g,x:names ⊢ fs_close1 B1[..,x] B2[..,x]] ⇒
          let ex_bstep [g,x:names ⊢ B2[..,x]] [g ⊢ B2'] e1 e2 =
          strengthen_bstep [g,x:names ⊢ B2[..,x]] in
          let [g,x:names ⊢ brefl] = e1 in
          let [g,x:names,z:names ⊢ prefl] = e2 in
          [g ⊢ fsc (fs_close1 (bs_res \x.B1[..,x]) B2')
          (c_trans sc_ext_res (c_res \w.(c_sym sc_ext_par)))]
     )
   | [g ⊢ c_sym C'] ⇒ let [g ⊢ D] = cong_fstepleft_impl_fstepright
     [g ⊢ C'] f in [g ⊢ D]
```

Figure 3.9: Proof of Lemma 2.6 (second part).

bound transition. We proceed by pattern matching on the contextual object `[g,x:names ⊢ F[..,x]]`, which represents a transition from $(P \mid Q)$ through an action α whose names do not include x. We consider two of the corresponding subcases:

- In the `fs_com1` case, `F` takes the form `[g,x:names ⊢ fs_com1 F1[..,x] B1[..,x]]`. Here, `F1` has type `[g,x:names ⊢ fstep P (f_out Z W) P']` and denotes a free output transition from P, while `B1` has type `[g,x:names ⊢ bstep Q[..] (b_in Z) \y.Q'[..,x,y]]` and denotes an input transition from Q. To eliminate the dependency on the name `x` from `B1` and obtain a transition `B1'` defined in the stronger context `g`, we apply the function `strengthen_bstep` to `B1` and unfold the objects `e1` and `e2` encoding equality. Next, we give an explicit type annotation to the object `F1`, in order to assign the explicit name `W` to the output name. Finally, we perform pattern matching on the output name `W`, which is defined in the context `(g,x:names)`: it can either be equal to x or be another variable occuring in the context `g`. Analogously to the informal proof, we conclude by exhibiting a different transition and congruence based on the case considered. We note that the syntax `#w` denotes parameter variables, i.e. variables occurring within a context; in this situation, a parameter variable is used to assert that `W` occurs within the context `g`.

- In the `fs_close` case, the proof is similar to the previous case: we apply the function `strengthen_bstep` to obtain a transition defined in a stronger context and then conclude. Unlike the informal proof, additional lemmas are not required and side conditions are automatically verified.

– In the `c_sym` case we conclude by recursively calling the symmetrical function `cong_fstepleft_impl_stepright`.

□

We provide the proof of the lemma `cong_bstepleft_impl_stepright`, which proves the "left" assertion for bound transitions.

```
and rec cong_bstepleft_impl_bstepright: (g:ctx) [g ⊢ P cong Q]
  → [g ⊢ bstep P A \x.P'[..,x]] → [g ⊢ ex_bstepcong P Q A \x.P'[..,x]] =
fn c ⇒ fn b ⇒ case c of
    | [g ⊢ sc_ext_par] ⇒
      (case b of
         | [g ⊢ bs_par2 B2] ⇒
           [g ⊢ bsc (bs_res \x.(bs_par2 B2[..])) \x.sc_ext_par]
      )
    | [g ⊢ c_par C1] ⇒
      (case b of
         | [g ⊢ bs_par2 B1] ⇒
           [g ⊢ bsc (bs_par2 B1) \x.(c_par C1[..])]
      )
```

Figure 3.10: Proof of Lemma 2.6 (third part).

Proof Verbalization: The proof proceeds by induction on the structure of the congruence `c`. The subcases `sc_ext_par` and `c_par` are proved analogously to their counterparts in the `cong_fstepleft_impl_stepright` function; together with the proof terms in the former function, they complete the encoding of the corresponding subcases of the informal proof. □

Lastly, we provide the proof of the lemma `cong_bstepright_impl_stepleft`, which proves the "right" assertion for bound transitions, in Figure 3.11.

Proof Verbalization: The proof proceeds by induction on the structure of the congruence `c`. Here, we only consider the `sc_ext_par` case, in which `c` represents the congruence $(\nu x)P \mid Q \equiv (\nu x)(P \mid Q)$ where x does not occur free in Q. We perform pattern matching on the bound variable `b`, representing a transition which starts from the process $(\nu x)(P \mid Q)$; here, we only consider the `[g ⊢ bs_open \x.F[..,x]]` case. The variable `F` denotes the transition $(P \mid Q) \xrightarrow{\bar{z}x} R$, with $x \neq z$. By pattern matching on `F` we obtain two subcases, `fs_par1` and `fs_par2`: analogously to the informal proof, in the first case we conclude by exhibiting the required congruence and transition, while in the second case we conclude through the `impossible` keyword due to a contradiction. □

```
and rec cong_bstepright_impl_bstepleft: (g:ctx) [g ⊢ P cong Q]
  → [g ⊢ bstep Q A \x.Q'[..,x]] → [g ⊢ ex_bstepcong Q P A \x.Q'[..,x]] =
fn c ⇒ fn b ⇒ case c of
   | [g ⊢ sc_ext_par] ⇒
     (case b of
        | [g ⊢ bs_open \x.F[..,x]] ⇒
          (case [g,x:names ⊢ F[..,x]] of
             | [g,x:names ⊢ fs_par1 F1[..,x]] ⇒
               [g ⊢ bsc (bs_par1 (bs_open \x.F1[..,x])) \x.c_ref]
             | [g,x:names ⊢ fs_par2 F2[..,x]] ⇒
               let ex_fstep [g,x:names ⊢ F2[..,x]] [g ⊢ F2'] e1 e2 =
               strengthen_fstep [g,x:names ⊢ F2[..,x]] in impossible e1
          )
     )
;
```

Figure 3.11: Proof of Lemma 2.6 (fourth part).

Lemma 2.7: `red_impl_red_rew`

We recall the statement of Lemma 2.7: "if $P \to Q$, then there exist three names x, y and z, a finite (possibly empty) set of names $w_1, ..., w_n$ and three processes R_1, R_2 and S such that $P \equiv (\nu w_1)...(\nu w_n)(\,(\bar{x}y.R_1 \mid x(z).R_2) \mid S\,)$ and $Q \equiv (\nu w_1)...(\nu w_n)(\,(R_1 \mid R_2\{y/z\}) \mid S\,)$."
Analogously to the encodings of lemmas 1.1, 1.2 and 1.3, we define a new type family `ex_red_rew` which encodes the existence of two congruences as stated in the conclusion of Lemma 2.7; in other words, it encodes the rewriting (up to congruence) of processes involved in a reduction. This type family has two constructors for the encoding of telescopes.

```
LF ex_red_rew: proc → proc → type =
  | red_base: P cong (((p_out X Y R1) p_par (p_in X R2)) p_par S)
    → Q cong ((R1 p_par (R2 Y)) p_par S) → ex_red_rew P Q
  | red_ind: P cong (p_res P') → Q cong (p_res Q')
    → ({w:names} ex_red_rew (P' w) (Q' w)) → ex_red_rew P Q
;
```

The type `ex_red_rew` is indexed by two processes. The first constructor `red_base` estabilishes this fact: if `P cong (((p_out X Y R1) p_par (p_in X R2)) p_par S)` holds for some processes `R1`, `S` and a process abstraction `R2`, and if `Q cong ((R1 p_par (R2 Y)) p_par S)` holds as well, then `ex_red_rew P Q` holds. The second constructor `red_ind` states that if P and Q are congruent to restrictions $(\nu w)P'$ and $(\nu w)Q'$ respectively, and `ex_red_rew (P' w) (Q' w)` holds for any choice of binder name w, then `ex_red_rew P Q` holds.

We present the proof of Lemma 2.7 in Figure 3.12.

Proof Verbalization: The proof proceeds by introducing the variable `r`, denoting the reduction `[g ⊢ P red Q]`, and performing pattern matching on it.

- In case `r` is obtained through the constructor `r_com`, we have that `P` and `Q` have the form `(p_out X Y R1) p_par (p_in X R2)` and `R1 p_par (R2 Y)`, respectively. We conclude by providing the congruences `P cong (((p_out X Y R1)`

```
rec red_impl_red_rew: (g:ctx) [g ⊢ P red Q] → [g ⊢ ex_red_rew P Q] =
/ total r (red_impl_red_rew _ _ _ r) /
fn r ⇒ case r of
  | [g ⊢ r_com] ⇒ [g ⊢ red_base (c_sym par_unit) (c_sym par_unit)]
  | [g ⊢ r_par R1]:[g ⊢ (P p_par R) red (Q p_par R)] ⇒
    let [g ⊢ D1] = red_impl_red_rew [g ⊢ R1] in
    let [g ⊢ D2] = red_impl_red_rew_par [g ⊢ R] [g ⊢ D1] in [g ⊢ D2]
  | [g ⊢ r_res \z.R1[..,z]] ⇒
    let [g,z:names ⊢ D1[..,z]] = red_impl_red_rew
    [g,z:names ⊢ R1[..,z]] in
    let [g ⊢ D2] = red_impl_red_rew_res [g,z:names ⊢ D1[..,z]] in
    [g ⊢ D2]
  | [g ⊢ r_str C1 R1 C2] ⇒
    let [g ⊢ D1] = red_impl_red_rew [g ⊢ R1] in
    let [g ⊢ D2] = red_impl_red_rew_str [g ⊢ C1] [g ⊢ D1] [g ⊢ C2] in
    [g ⊢ D2]
;
```

Figure 3.12: Proof of Lemma 2.7.

`p_par (p_in X R2)) p_par p_zero)` and `Q cong ((R1 p_par (R2 Y)) p_par p_zero)`, which are witnessed by the object `c_sym par_unit`, within the object `[g ⊢ red_base (c_sym par_unit) (c_sym par_unit)]`.

- In the other three cases, the proofs have the same structure: we recursively invoke the function `red_impl_red_rew` on a structurally smaller reduction `R1`, obtaining the corresponding object `D1` of type `ex_red_rew`; then, we use an additional lemma for each case, in order to unfold the object `D1` and build the required rewriting of processes.

□

We now present the proof of the auxiliary lemma `red_impl_red_rew_par` in Figure 3.13, omitting details regarding the other two auxiliary lemmas. In mathematical terms, it corresponds to the following result: "given two processes P and Q such that $P \equiv (\overline{\nu w})((\bar{x}y.R_1 \mid x(z).R_2) \mid S)$ and $Q \equiv (\overline{\nu w})((R_1 \mid R_2\{y/z\}) \mid S)$ for some R_1, R_2, S and $\overline{w}$, then it can be demonstrated that, for any process R in which each w_i does not occur free, $P \mid R \equiv (\overline{\nu w})((\bar{x}y.R_1' \mid x(z).R_2') \mid S')$ and $Q \mid R \equiv (\overline{\nu w})((R_1' \mid R_2'\{y/z\}) \mid S')$ for some processes R_1', R_2' and S'". In our case, we are going to show that $R_1' := R_1$, $R_2' := R_2$ and $S' := S \mid R$.

Proof Verbalization: The proof proceeds by performing pattern matching on the variable `d`.

- In case `d` has the form `[g ⊢ red_base C1 C2]`, the objects `C1` and `C2` represent the congruences $P \equiv (\bar{x}y.R_1 \mid x(z).R_2) \mid S$ and $Q \equiv (R_1 \mid R_2\{y/z\}) \mid S$. We immediately conclude by providing the appropriate congruences of the processes $P \mid R$ and $Q \mid R$.

- If `d` has the form `[g ⊢ red_ind C1 C2 \w.D1[..,w]]`, `C1` and `C2` denote objects of type `[g ⊢ P cong (p_res P')]` and `[g ⊢ Q cong (p_res Q')]`, while

```
rec red_impl_red_rew_par: (g:ctx) {R:[g ⊢ proc]} [g ⊢ ex_red_rew P Q]
  → [g ⊢ ex_red_rew (P p_par R) (Q p_par R)] =
/ total d (red_impl_red_rew_par _ _ _ _ d) /
mlam R ⇒ fn d ⇒ case d of
   | [g ⊢ red_base C1 C2] ⇒
     [g ⊢ red_base (c_trans (c_par C1) (c_sym par_assoc))
     (c_trans (c_par C2) (c_sym par_assoc))]
   | [g ⊢ red_ind C1 C2 \w.D1[..,w]] ⇒
     let [g,w:names ⊢ D2[..,w]] = red_impl_red_rew_par
     [g,w:names ⊢ R[..]] [g,w:names ⊢ D1[..,w]] in
     [g ⊢ red_ind (c_trans (c_par C1) sc_ext_par)
     (c_trans (c_par C2) sc_ext_par) \w.D2[..,w]]
;
```

Figure 3.13: Proof of the auxiliary lemma for Lemma 2.7

D1 denotes an object of type `[g,w:names ⊢ ex_red_rew (P’ w) (Q’ w)]`. We recursively apply the function red_impl_red_rew_par to the arguments `R[..]`, D1 in the weaker context `(g,w:names)`, obtaining a parametric derivation D2 of `[g,w:names ⊢ ex_red_rew ((P’ w) p_par R[..]) ((Q’ w) p_par R[..])]`. Finally, we conclude by passing the appropriate congruences and the object D2 to the `red_ind` constructor.

□

Theorem 2: `red_impl_fstepcong`

We provide the proof encoding for Theorem 2, which states that $P \to Q$ implies the existence of a Q' such that $P \xrightarrow{\tau} Q'$ and $Q \equiv Q'$.

```
rec red_impl_fstepcong: (g:ctx) [g ⊢ P red Q]
  → [g ⊢ ex_fstepcong P P f_tau Q] =
/ total r (red_impl_fstepcong _ _ _ r) /
fn r ⇒ let [g ⊢ D1] = red_impl_red_rew r in
       let [g ⊢ D2] = red_rew_impl_fstepcong [g ⊢ D1] in [g ⊢ D2]
;
```

Proof Verbalization: We introduce the variable r, which represents the reduction $P \to Q$. Then, we apply the function red_impl_red_rew on r, returning an object D1 of type `[g ⊢ ex_red_rew P Q]`, which encodes the following congruences for some R_1, R_2, S and $\overline{w}$: $P \equiv (\nu w_1)...(\nu w_n)((\bar{x}y.R_1 \mid x(z).R_2) \mid S)$ and $Q \equiv (\nu w_1)...(\nu w_n)((R_1 \mid R_2\{y/z\}) \mid S)$. Finally, we invoke the auxiliary function red_rew_impl_fstepcong, which unfolds the argument D1 and returns the desired object of type `[g ⊢ ex_fstepcong P P f_tau Q]`. □

We now present the encoding of the auxiliary lemma used in the former proof, implemented by the recursive function red_rew_impl_fstepcong. In mathematical terms, it corresponds to demonstrating the following result: “given the congruences $P \equiv (\overline{\nu w})((\bar{x}y.R_1 \mid x(z).R_2) \mid S)$ and $Q \equiv (\overline{\nu w})((R_1 \mid R_2\{y/z\}) \mid S)$ for some R_1, R_2, S and $\overline{w}$, then there exists a process Q' such that $P \xrightarrow{\tau} Q'$ and $Q \equiv Q'$”.

```
rec red_rew_impl_fstepcong: (g:ctx) [g ⊢ ex_red_rew P Q]
  → [g ⊢ ex_fstepcong P P f_tau Q] =
/ total d (red_rew_impl_fstepcong _ _ _ d) /
fn d ⇒ case d of
   | [g ⊢ red_base C1 C2] ⇒
     let [g ⊢ fsc F C3] = cong_fstepright_impl_fstepleft
     [g ⊢ C1] [g ⊢ fs_par1 (fs_com1 fs_out bs_in)] in
     [g ⊢ fsc F (c_trans C2 C3)]
   | [g ⊢ red_ind C1 C2 \w.D1[..,w]] ⇒
     let [g,w:names ⊢ fsc F1[..,w] C3[..,w]] =
     red_rew_impl_fstepcong [g,w:names ⊢ D1[..,w]] in
     let [g ⊢ fsc F2 C4] = cong_fstepright_impl_fstepleft
     [g ⊢ C1] [g ⊢ fs_res \w.F1[..,w]] in
     [g ⊢ fsc F2 (c_trans C2 (c_trans (c_res \w.C3[..,w]) C4))]
;
```

Proof Verbalization: The proof proceeds by performing pattern matching on the variable `d`.

– If `d` has the form `[g ⊢ red_base C1 C2]`, the objects `C1` and `C2` denote congruences of type `[g ⊢ P cong (((p_out X Y R1) p_par (p_in X R2)) p_par S)]` and `[g ⊢ Q cong ((R1 p_par (R2 Y)) p_par S)]` respectively. For ease of notation, we set $P \equiv P''$ and $Q \equiv Q''$ for the congruences above. We apply the function `cong_fstepright_impl_fstepleft` on the arguments `C1`, representing the congruence $P \equiv P''$, and `(fs_par1 (fs_com1 fs_out bs_in))` ($\star$), representing the transition $P'' \xrightarrow{\tau} Q''$; as a result, we obtain the objects `F` and `C3`, respectively denoting the transition $P \xrightarrow{\tau} Q'$ and the congruence $Q'' \equiv Q'$, as illustrated in the diagram below. Finally, we conclude by exhibiting the transition `F` and the congruence `c_trans C2 C3`, which states that $Q \equiv Q'$.

$$\begin{tikzcd}
P \arrow[d] & & P \arrow[d, dashed, "\texttt{F}"', "\tau"] \arrow[r, phantom, "\overset{\texttt{C1}}{\equiv}"] & P'' \arrow[d, "\star"', "\tau"] & \\
Q \arrow[rr, phantom, "\Rightarrow"] & & Q' \arrow[r, phantom, "\overset{\texttt{C3}}{\equiv}"] & Q'' \arrow[r, phantom, "\overset{\texttt{C2}}{\equiv}"] & Q
\end{tikzcd}$$

Figure 3.14: Graphical representation of the `red_base` case in the former proof.

– If `d` has the form `[g ⊢ red_ind C1 C2 \w.D1[..,w]]`, `C1` and `C2` denote congruences of type `[g ⊢ P cong (p_res P'')]` and `[g ⊢ Q cong (p_res Q'')]`, while `D1` has type `[g,w:names ⊢ ex_red_rew (P'' w) (Q'' w)]`. By recursively applying the function `red_rew_impl_fstepcong` on `D1` in the weaker context `(g,w:names)`, we obtain an object of type `ex_fstepcong` of the form `fsc F1 C3`: the variable `F1` denotes a transition of type `[g,w:names ⊢ fstep (P'' w) f_tau (R'' w)]`, while the variable `C3` denotes a congruence of type `[g,w:names ⊢ (Q'' w) cong (R'' w)]`. Thus, the term `(fs_res \w.F1[..,w])` ($\star$) denotes a transition `[g ⊢ fstep (p_res P'') f_tau (p_res R'')]`. By applying the function `cong_fstepright_impl_fstepleft` on the transition ($\star$) and the congruence `C1`, we obtain the objects `F2` and `C4`: the former denotes a transition [g ⊢ fstep P f_tau Q'], while the latter denotes a congruence [g ⊢ (p_res R'') cong Q']. Finally, we provide the required transition and congruence, as displayed in the following diagram.

$$\begin{tikzcd}
P \arrow[d] & & P \arrow[d, dashed, "\texttt{F2}"', "\tau"] & \overset{\texttt{C1}}{\equiv} & (\nu w)P'' \arrow[d, "\star"', "\tau"] & & & & \\
Q \arrow[r, phantom, "\Rightarrow" near end, shift left=6ex] & & Q' & \overset{\texttt{C4}}{\vdots\!\equiv} & (\nu w)R'' & \overset{\texttt{c\_res C3}}{\equiv} & (\nu w)Q'' & \overset{\texttt{C2}}{\equiv} & Q
\end{tikzcd}$$

Figure 3.15: Graphical representation of the red_ind case in the former proof.

□

Chapter 4

Conclusions

4.1 Evaluation

In this thesis, we provide a formalization of two operational semantics for a fragment of the π-calculus and their equivalence in Beluga. Our implementation is organized in 2 theorems supported by 32 lemmas; it contains less than 100 lines of code for definitions and less than 600 lines for proofs. In Table 4.1 we exhibit a summary of definitions and proofs, together with their respective name in the Beluga encoding; Figures 4.1 and 4.2 display the dependency tree of the two theorems. Note that the tree in Figure 4.1 is split into two parts due to space constraints: the `red_rew_impl_fstepcong` node is intended to be connected to the `cong_fstepright_impl_fstepleft` node. The complete formalization is accessible at `https://github.com/GabrieleCecilia/concurrent-benchmark-solution`. The code presented in this thesis is directly taken from the executables available there.

The choice of the proof assistant Beluga offers several benefits. The HOAS encoding eliminates the need to state and prove technical lemmas regarding α-renaming and substitution. Additionally, this technique leads to a simplification of definitions and proofs, since there is no need to address side conditions about semantics rules. We also observe that the formal proof does not require lemmas about free and bound variables, unlike the informal one; the only technical lemma needed is the strengthening lemma, common in HOAS encodings, which can be seen as the counterpart of a lemma present in the informal proof.
A significant result is the one-to-one correspondence between informal and formal proof, apart from some duplication of definitions and theorems; as previously mentioned, this correspondence is displayed in Table 4.1. Each lemma and theorem corresponds to a recursive function; application of the inductive hypothesis to an argument matches to the recursive call of the function on the respective argument; case analysis is encoded through pattern matching. A noteworthy example of this correspondence occurs in Lemma 2.6, which includes a case distinction based on the equality of names; this translates into pattern matching over a name defined in a certain context, with the use of parameter variables to encode inequality. Despite the computational complexity that follows from the use of mutually recursive predicates and functions, the totality checker consistently ensures that each recursive function represents a proof. Although an unknown coverage error occurred in one instance, this issue was resolved by modifying a predicate definition.

One drawback of the HOAS encoding for the LTS semantics consists in the definition of two distinct types for free and bound transitions, which results in the

Definitions and Proofs	**Name in Beluga**	**Sections**
Names, processes	`names`, `proc`	2.1, 3.1
Congruence, reduction	`cong`, `red`	2.2.1, 3.2.1
Actions, transitions	`f_act`, `b_act`, `fstep`, `bstep`	2.2.2, 3.2.2
Rewriting of processes involved in input transitions: Lemma 1.1	`bs_in_rew`	2.3.1, 3.3.1
Rewriting of processes involved in free output transitions: Lemma 1.2	`fs_out_rew`	2.3.1, 3.3.1
Rewriting of processes involved in bound output transitions: Lemma 1.3	`bs_out_rew`	2.3.1, 3.3.1
τ-transition implies reduction: Theorem 1	`fstep_impl_red`	2.3.1, 3.3.1
Auxiliary lemmas about free/bound names in processes/actions: Lemma 2.1 to 2.5	`strengthen_fstep`, `strengthen_bstep`	2.3.2, 3.3.2
Congruence and left transition implies right transition and congruence: Lemma 2.6	`cong_fstepleft_impl_fstepright`, `cong_fstepright_impl_fstepleft`, `cong_bstepright_impl_bstepleft`, `cong_fstepleft_impl_fstepright`	2.3.2, 3.3.2
Rewriting of processes involved in reductions: Lemma 2.7	`red_impl_red_rew`	2.3.2, 3.3.2
Reduction implies τ-transition and congruence: Theorem 2	`red_impl_fstepcong`	2.3.2, 3.3.2

Table 4.1: Correspondence between informal and formal definitions and proofs.

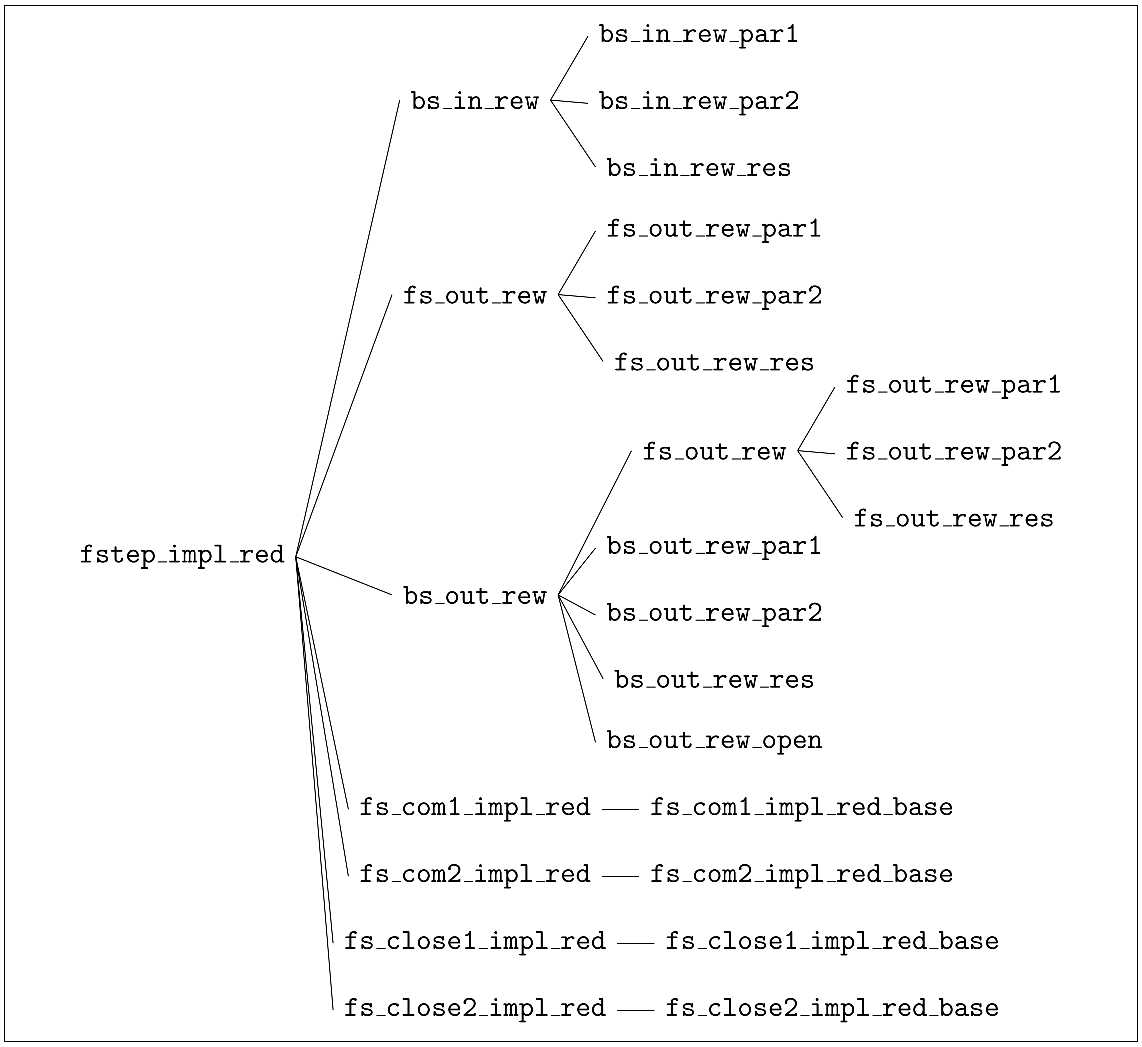

Figure 4.1: Dependency tree of the `fstep_impl_red` theorem.

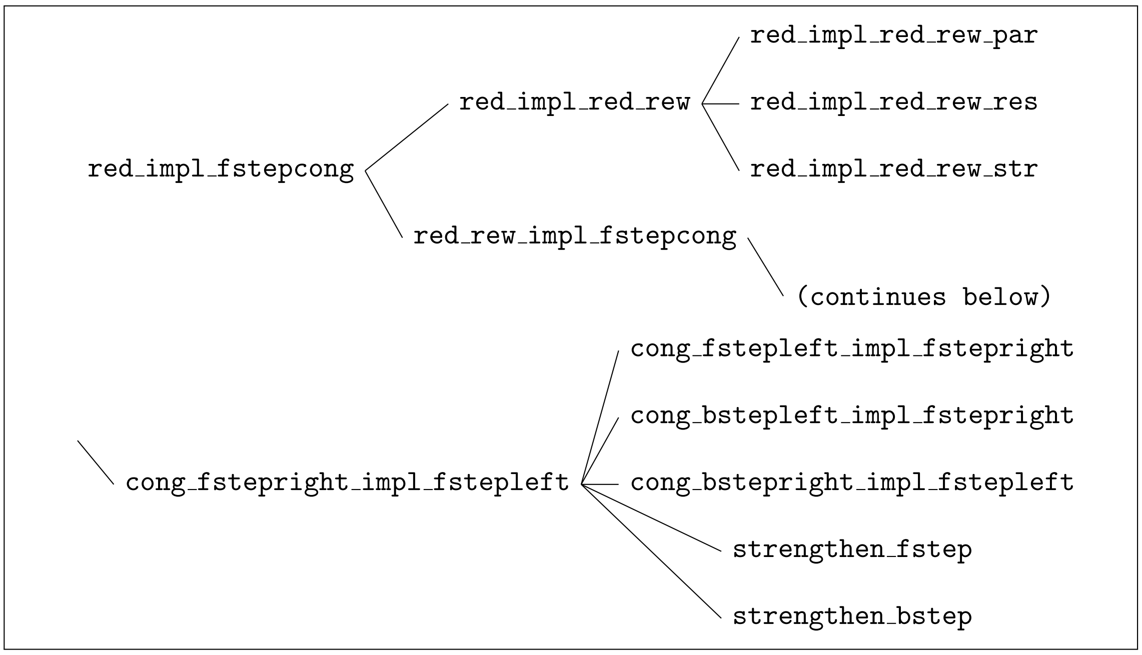

Figure 4.2: Dependency tree of the `red_impl_fstepcong` theorem.

duplication of some lemmas into pairs of mutual recursive functions. Moreover, the impossibility to prove conjunctions of statements in Beluga (such as in double implications or situations with symmetric predicates) leads to the duplication of lemmas regarding congruence. Additionally, the absence of a construct for existentials requires the definition of new type families to encode them. Overall, we believe that the advantages of employing Beluga outweigh these disadvantages.

Although the second challenge of the Concurrent Calculi Formalization Benchmark focuses on a subset of the π-calculus without some process constructs, we believe that our work holds practical utility for several reasons.
In the first place, both the informal and formal proof provided in this thesis are new. The equivalence between LTS and reduction semantics is a well-known result in the theory of π-calculus, and sources like Sangiorgi & Walker [23] already outline the main steps of its proof; however, as far as we know, there is no complete informal or formal proof of this theorem in the literature. While this result is generally accepted and undisputed for the standard π-calculus, the same cannot be said for extensions of the calculus, or when a new notion of bisimilarity is introduced; in such cases, the equivalence of the semantics might be subject to discussion.
Next, this thesis introduces a custom technique for encoding telescopic sequences of binders in processes and proving results about them; this approach also eliminates the need to define normalized derivations for reductions. Given a reduction $r : P \to Q$, a normalized derivation of r is a derivation in which reduction rules are applied in a specific order: namely, the first rule applied is R-Com, followed by a R-Par, a finite (potentially empty) series of R-Res, and finally a R-Struct. This notion is introduced in [23], where it is proved that every sequence of reductions has a normalized derivation; as a consequence, the same rewriting of processes involved in a reduction described in Lemma 2.7 can be obtained. In the previous sections we illustrated how this result can be obtained directly through an induction over the reduction's structure, and how the formalization is made possible by the encoding technique for telescopes mentioned earlier.
Additionally, our work provides an encoding technique for performing induction on multiple arguments in Beluga, as explained on page 40. Finally, being a solution to the second challenge of the Concurrent Benchmark, it contributes to the achievement of the objectives set by the challenge authors: specifically, it can serve for the comparison of encoding techniques for scope extrusion in different proof environments and it can lay the groundwork for future projects in this field.

4.2 Related and Future Work

Several works offer formalizations of the π-calculus operational semantics with different approaches and techniques. Here, we mention the HOAS encodings of Honsell et al. [10] and Miller & Tiu [24]. Honsell's formalization implements LTS semantics for a more comprehensive version of the π-calculus and focuses on encoding the theory of strong late bisimilarity. Honsell's approach includes the definition of auxiliary predicates to encode name freshness at the object level, which allow reasoning about bisimilarity and dealing with inequality of names for the mismatch operator. Conversely, Miller & Tiu present a formalization of the π-calculus within a logic that contains the ∇ operator, which is used to encode quantification over fresh names. Their formalization addresses the use of this quantifier to encode generic

judgements and covers the differences between open and late bisimulation, which arise from the differences between the $\forall$ and ∇ operators. Another difference from Honsell's work is the encoding of actions: instead of defining two type families for free and bound actions, Miller & Tiu provide one type with three constructors (for input, output and τ actions). We have proved that the two approaches, when restricted to the subset of the π-calculus introduced in the second challenge of the concurrent benchmark, are equivalent: a formalization of this result is available at `https://github.com/GabrieleCecilia/concurrent-benchmark-solution`.
Our formalization takes inspiration from both prior works, particularly Honsell's approach to encoding LTS semantics, while developing and proving new results. Furthermore, we observe that our formalization does not require the deployment of techniques to encode freshness of names, such as the aforementioned auxiliary predicates or ∇ operator. This is because the calculus defined in the second challenge of the benchmark does not include the mismatch operator, requires contexts with a very simple structure, and the challenge does not involve bisimilarity.

In the future, a potential direction of development consists in extending the syntax of processes to include sums, match operator and replication, and adjusting the semantics rules accordingly. Introducing sums, for example, requires updating the definitions of the predicates encoding rewritings of processes involved in transitions or reductions; as a consequence, some theorem proofs become more complicated. As for replication, the literature does not provide a uniform definition of transition and congruence axioms (see [23] or [9] for reference), and additional complexity arises due to the infinite nature of process behaviours. Including the mismatch operator requires addressing issues related to the encoding of name inequality, which is typically challenging in LF; while Honsell's approach offers a viable solution, its compatibility with Beluga remains to be verified.
Another avenue for improvement lies in the implementation of these results in Harpoon [6], an interactive prover for Beluga. Employing tactics and automation could streamline the proof search and enhance understanding of proof strategies.
Lastly, it is worth noting that while the first and second challenges of the Concurrent Calculi Formalization Benchmark have been addressed, progress remains to be made on the third challenge regarding coinduction in the π-calculus.

Declaration

I declare that I have used ChatGPT solely for the purpose of correcting grammar and refining text wording in this thesis. Furthermore, I confirm that I have not used ChatGPT for any other purpose and declare that I wrote this thesis independently, without any other external assistance, except for the quoted literature.

Ringraziamenti

Ringrazio il mio relatore, Alberto Momigliano, che con notevole dedizione, professionalità, cura e pazienza mi ha orientato e seguito nella stesura di questa tesi. Le qualità che ha dimostrato in questi mesi, dal punto di vista sia tecnico che umano, non sono per nulla scontate e meritano sincera riconoscenza.

Ringrazio la mia famiglia, che è sempre stata al mio fianco e in questi anni mi ha sempre messo nelle condizioni migliori affinché potessi condurre con successo e serenità il percorso universitario.

Ringrazio i miei amici: Marco, Pietro e Mattia, che sono stati particolarmente presenti nei momenti di difficoltà; Filippo, Franco, Lorenzo e Tullio, a cui devo molti dei momenti di gioia vissuti in dipartimento; gli amici del coro Capitano Grandi e della Schola Cantorum, perché insieme abbiamo condiviso molto e insieme siamo cresciuti.

Appendix A

Equivalence between Early and Late Semantics

In order to formalize equivalence between the two semantics, we first need to encode the semantics rules in the same environment. While the encodings of names and processes remain consistent, we add a constructor `f_in` to the type family `f_act` to represent free output actions in the early semantics.

```
LF f_act: type =
  | f_in: names → names → f_act
  | f_out: names → names → f_act
  | f_tau: f_act
;

LF b_act: type =
  | b_in: names → b_act
  | b_out: names → b_act
;
```

Figure A.1: Encoding of the set of actions.

Next we define the type families `early_fstep` and `early_bstep`, which represent free and bound transitions in the early semantics, in Figure A.2.
Compared to the late semantics, we introduced a new constructor `efs_in` and modified the constructors `efs_com1` and `efs_com2`. The `efs_in` constructor represents the transition $x(z).P \xrightarrow{x(y)} P\{y/z\}$ for all inputs y. The `efs_com` constructors represent the communication of processes within a parallel composition only when the input and output names are identical. We observe that the `ebs_in` constructor representing bound input transitions cannot be eliminated, since bound input transitions are needed as premises in the rules introduced by the `efs_close` constructors.

For uniformity of notation, we modified the name of the `fstep` and `bstep` types, encoding transitions in the late semantics, into `late_fstep` and `late_bstep`. The names of their constructors are adjusted accordingly. This modification does not alter the meaning of the rules defined in Figure 3.5. We refrain from providing the new type family definitions for the sake of brevity.

```
LF early_fstep: proc → f_act → proc → type =
  | efs_in: early_fstep (p_in X P) (f_in X Y) (P Y)
  | efs_out: early_fstep (p_out X Y P) (f_out X Y) P
  | efs_par1: early_fstep P A P' → early_fstep (P p_par Q) A (P' p_par Q)
  | efs_par2: early_fstep Q A Q' → early_fstep (P p_par Q) A (P p_par Q')
  | efs_com1: early_fstep P (f_out X Y) P' → early_fstep Q (f_in X Y) Q'
    → early_fstep (P p_par Q) f_tau (P' p_par Q')
  | efs_com2: early_fstep P (f_in X Y) P' → early_fstep Q (f_out X Y) Q'
    → early_fstep (P p_par Q) f_tau (P' p_par Q')
  | efs_res: ({z:names} early_fstep (P z) A (P' z))
    → early_fstep (p_res P) A (p_res P')
  | efs_close1: early_bstep P (b_out X) P' → early_bstep Q (b_in X) Q'
    → early_fstep (P p_par Q) f_tau (p_res \z.((P' z) p_par (Q' z)))
  | efs_close2: early_bstep P (b_in X) P' → early_bstep Q (b_out X) Q'
    → early_fstep (P p_par Q) f_tau (p_res \z.((P' z) p_par (Q' z)))

and early_bstep: proc → b_act → (names → proc) → type =
  | ebs_in: early_bstep (p_in X P) (b_in X) P
  | ebs_par1: early_bstep P A P'
    → early_bstep (P p_par Q) A \x.((P' x) p_par Q)
  | ebs_par2: early_bstep Q A Q'
    → early_bstep (P p_par Q) A \x.(P p_par (Q' x))
  | ebs_res: ({z:names} early_bstep (P z) A (P' z))
    → early_bstep (p_res P) A \x.(p_res \z.(P' z x))
  | ebs_open: ({z:names} early_fstep (P z) (f_out X z) (P' z))
    → early_bstep (p_res P) (b_out X) P'
;
```

Figure A.2: Encoding of the early LTS semantics rules.

We prove the equivalence of semantics following Parrow's approach [16]. Namely, our objective is to demonstrate that "the τ-transitions that can be inferred with the early semantics are exactly those that can be inferred with the late semantics". In order to achieve the result by induction on the depth of the inference of the transitions, we need some additional lemmas stating that "not only τ-actions but also input and output actions correspond in the two semantics", where the correspondence between input and free input is defined as follows: $P \xrightarrow{x(y)} P'$ holds in the early semantics if and only if there exist a process P'' and a name w such that $P \xrightarrow{x(w)} P''$ holds in the late semantics and $P' = P''\{y/w\}$. $(\star)$

We begin with the definition of a type family ex_latebs which encodes the existence of a late transition such as in the correspondence $(\star)$:

```
LF ex_latebs: proc → names → names → proc → type =
  | lbs: late_bstep P (b_in X) \w.(Q' w) → eqp Q (Q' Y)
    → ex_latebs P X Y Q
;
```

The constructor lbs asserts that, given two processes P and Q and two names X and Y, ex_latebs P X Y Q holds if there exists a process abstraction Q' satisfying the following conditions: late_bstep P (b_in X) \w.(Q' w) holds, and Q' Y = Q.

We present the proofs of the two lemmas encoding the correspondence between free input transitions in the early semantics and input transitions in the late semantics. The `finp_earlytolate` lemma encodes one implication and the `finp_latetoearly` lemma encodes the other implication.

```
rec finp_earlytolate: (g:ctx) [g ⊢ early_fstep P (f_in X Y) Q]
  → [g ⊢ ex_latebs P X Y Q] =
/ total e (finp_earlytolate _ _ _ _ _ e) /
fn e ⇒ case e of
  | [g ⊢ efs_in] ⇒ [g ⊢ lbs lbs_in prefl]
  | [g ⊢ efs_par1 FE1] ⇒
    let [g ⊢ lbs BL1 prefl] = finp_earlytolate [g ⊢ FE1] in
    [g ⊢ lbs (lbs_par1 BL1) prefl]
  | [g ⊢ efs_par2 FE2] ⇒
    let [g ⊢ lbs BL2 prefl] = finp_earlytolate [g ⊢ FE2] in
    [g ⊢ lbs (lbs_par2 BL2) prefl]
  | [g ⊢ efs_res \x.FE1[..,x]] ⇒ let [g,x:names ⊢ lbs BL1[..,x] prefl]
    = finp_earlytolate [g,x:names ⊢ FE1[..,x]] in
    [g ⊢ lbs (lbs_res \x.BL1[..,x]) prefl]
;
```

Proof Verbalization: We assume the hypothesis `e`, whose type is `[g ⊢ early_fstep P (f_in X Y) Q]`, and conduct case analysis on `e`.
In case it was obtained through the constructor `efs_in`, it represents an object of type `[g ⊢ early_fstep (p_in X Q) (f_in X Y) (Q Y)]`; since the input prefix `p_in X Q` is able to perform a late input transition through the constructor `lbs_in` to the process `Q`, which equals to `(Q Y)` when applied to `Y`, we reached the conclusion.
In the other cases, we perform a recursive call on a structurally smaller free input transition, unfolding the result by stating it was built with the constructor `lbs` and collecting the transition required to conclude immediately. □

```
rec finp_latetoearly: (g:ctx) {Y:[g ⊢ names]} [g ⊢ ex_latebs P X Y Q]
  → [g ⊢ early_fstep P (f_in X Y) Q] =
/ total p (finp_latetoearly _ _ p _ _ _) /
mlam Y ⇒ fn l ⇒ let [g ⊢ lbs L prefl] = l in
case [g ⊢ L] of
  | [g ⊢ lbs_in] ⇒ [g ⊢ efs_in]
  | [g ⊢ lbs_par1 BL1] ⇒ let [g ⊢ FE1] =
    finp_latetoearly [g ⊢ Y] [g ⊢ lbs BL1 prefl] in [g ⊢ efs_par1 FE1]
  | [g ⊢ lbs_par2 BL2] ⇒ let [g ⊢ FE2] =
    finp_latetoearly [g ⊢ Y] [g ⊢ lbs BL2 prefl] in [g ⊢ efs_par2 FE2]
  | [g ⊢ lbs_res \x.BL1[..,x]] ⇒
    let [g,x:names ⊢ FE1[..,x]] = finp_latetoearly [g,x:names ⊢ Y[..]]
    [g,x:names ⊢ lbs (BL1[..,x]) prefl] in [g ⊢ efs_res \x.FE1[..,x]]
;
```

Proof Verbalization: In the statement of this lemma, the input name `Y` needs to be passed as an explicit argument: otherwise, Beluga would not be able to reconstruct it during some further calls of this lemma. After introducing the variables `Y` and `l`, the latter is unfolded using the `let` keyword, in order to obtain the variable `L` denoting the late transition and perform unification of the involved variables. After these steps, the proof proceeds analogously to the previous lemma. □

The correspondence between the other types of actions is straightforward. We provide the formalization of the free output case, omitting the description of the other cases.

```
rec fout_earlytolate: (g:ctx) [g ⊢ early_fstep P (f_out X Y) Q]
  → [g ⊢ late_fstep P (f_out X Y) Q] =
/ total e (fout_earlytolate _ _ _ _ _ e) /
fn e ⇒ case e of
  | [g ⊢ efs_out] ⇒ [g ⊢ lfs_out]
  | [g ⊢ efs_par1 FE1] ⇒
    let [g ⊢ FL1] = fout_earlytolate [g ⊢ FE1] in [g ⊢ lfs_par1 FL1]
  | [g ⊢ efs_par2 FE2] ⇒
    let [g ⊢ FL2] = fout_earlytolate [g ⊢ FE2] in [g ⊢ lfs_par2 FL2]
  | [g ⊢ efs_res \x.FE1[..,x]] ⇒
    let [g,x:names ⊢ FL1[..,x]]= fout_earlytolate[g,x:names ⊢ FE1[..,x]]
    in [g ⊢ lfs_res \x.FL1[..,x]]
;

rec fout_latetoearly: (g:ctx) [g ⊢ late_fstep P (f_out X Y) Q]
  → [g ⊢ early_fstep P (f_out X Y) Q] =
/ total l (fout_latetoearly _ _ _ _ _ l) /
fn l ⇒ case l of
  | [g ⊢ lfs_out] ⇒ [g ⊢ efs_out]
  | [g ⊢ lfs_par1 FL1] ⇒
    let [g ⊢ FE1] = fout_latetoearly [g ⊢ FL1] in [g ⊢ efs_par1 FE1]
  | [g ⊢ lfs_par2 FL2] ⇒
    let [g ⊢ FE2] = fout_latetoearly [g ⊢ FL2] in [g ⊢ efs_par2 FE2]
  | [g ⊢ lfs_res \x.FL1[..,x]] ⇒
    let [g,x:names ⊢ FE1[..,x]]= fout_latetoearly[g,x:names ⊢ FL1[..,x]]
    in [g ⊢ efs_res \x.FE1[..,x]]
;
```

Figure A.3: Proof of the lemmas regarding correspondence of free output transitions.

Finally, we are ready to state and prove the theorems about equivalence of semantics. The `tau_earlytolate` theorem states that any τ-transition in the early semantics corresponds to a τ-transition in the late semantics; the `tau_latetoearly` states the other implication. We begin with the first theorem; the proof is in Figure A.4.

Proof Verbalization: We introduce the variable `e` representing the early τ-transition and conduct case analysis on its structure. In the `efs_par` cases, a recursive call of the `tau_earlytolate` function is sufficient to conclude; the same holds for the `efs_res` case. In the `efs_com` and `efs_close` cases, we have some specific early transitions as hypotheses: by applying the lemmas stated above, we obtain the corresponding late transitions and conclude. □

```
rec tau_earlytolate: (g:ctx) [g ⊢ early_fstep P f_tau Q]
  → [g ⊢ late_fstep P f_tau Q] =
/ total e (tau_earlytolate _ _ _ e) /
fn e ⇒ case e of
  | [g ⊢ efs_par1 FE1] ⇒
    let [g ⊢ FL1] = tau_earlytolate [g ⊢ FE1] in [g ⊢ lfs_par1 FL1]
  | [g ⊢ efs_par2 FE2] ⇒
    let [g ⊢ FL2] = tau_earlytolate [g ⊢ FE2] in [g ⊢ lfs_par2 FL2]
  | [g ⊢ efs_com1 FE1 FE2] ⇒
    let [g ⊢ FL1] = fout_earlytolate [g ⊢ FE1] in
    let [g ⊢ lbs BL2 prefl] = finp_earlytolate [g ⊢ FE2] in
    [g ⊢ lfs_com1 FL1 BL2]
  | [g ⊢ efs_com2 FE1 FE2] ⇒
    let [g ⊢ lbs BL1 prefl] = finp_earlytolate [g ⊢ FE1] in
    let [g ⊢ FL2] = fout_earlytolate [g ⊢ FE2] in
    [g ⊢ lfs_com2 BL1 FL2]
  | [g ⊢ efs_res \x.FE1[..,x]] ⇒
    let [g,x:names ⊢ FL1[..,x]] = tau_earlytolate [g,x:names ⊢ FE1[..,x]]
    in [g ⊢ lfs_res \x.FL1[..,x]]
  | [g ⊢ efs_close1 BE1 BE2] ⇒
    let [g ⊢ BL1] = bout_earlytolate [g ⊢ BE1] in
    let [g ⊢ BL2] = binp_earlytolate [g ⊢ BE2] in
    [g ⊢ lfs_close1 BL1 BL2]
  | [g ⊢ efs_close2 BE1 BE2] ⇒
    let [g ⊢ BL1] = binp_earlytolate [g ⊢ BE1] in
    let [g ⊢ BL2] = bout_earlytolate [g ⊢ BE2] in
    [g ⊢ lfs_close2 BL1 BL2]
;
```

Figure A.4: Proof of the first equivalence theorem.

The proof of the second theorem is provided in Figure A.5.

Proof Verbalization: The proof has a similar structure to the previous one. We observe that, in the `lfs_com` cases, we need to introduce a typing annotation to the objects `FE1` and `FE2` which represent the early free output transitions. This allows us to assign the name `Y` to the variable representing the exchanged name in the process interaction; as a result, we are now able to pass it as an argument to the `finp_latetoearly` function and conclude. □

```
rec tau_latetoearly: (g:ctx) [g ⊢ late_fstep P f_tau Q]
  → [g ⊢ early_fstep P f_tau Q] =
/ total l (tau_latetoearly _ _ _ l) /
fn l ⇒ case l of
  | [g ⊢ lfs_par1 FL1] ⇒
    let [g ⊢ FE1] = tau_latetoearly [g ⊢ FL1] in [g ⊢ efs_par1 FE1]
  | [g ⊢ lfs_par2 FL2] ⇒
    let [g ⊢ FE2] = tau_latetoearly [g ⊢ FL2] in [g ⊢ efs_par2 FE2]
  | [g ⊢ lfs_com1 FL1 BL2] ⇒
    let [g ⊢ FE1]:[g ⊢ early_fstep P1 (f_out X Y) Q1] =
    fout_latetoearly [g ⊢ FL1] in
    let [g ⊢ FE2] = finp_latetoearly [g ⊢ Y] [g ⊢ lbs BL2 prefl] in
    [g ⊢ efs_com1 FE1 FE2]
  | [g ⊢ lfs_com2 BL1 FL2] ⇒
    let [g ⊢ FE2]:[g ⊢ early_fstep P2 (f_out X Y) Q2] =
    fout_latetoearly [g ⊢ FL2] in
    let [g ⊢ FE1] = finp_latetoearly [g ⊢ Y] [g ⊢ lbs BL1 prefl] in
    [g ⊢ efs_com2 FE1 FE2]
  | [g ⊢ lfs_res \x.FL1[..,x]] ⇒
    let [g,x:names ⊢ FE1[..,x]] = tau_latetoearly
    [g,x:names ⊢ FL1[..,x]] in [g ⊢ efs_res \x.FE1[..,x]]
  | [g ⊢ lfs_close1 BL1 BL2] ⇒
    let [g ⊢ BE1] = bout_latetoearly [g ⊢ BL1] in
    let [g ⊢ BE2] = binp_latetoearly [g ⊢ BL2] in
    [g ⊢ efs_close1 BE1 BE2]
  | [g ⊢ lfs_close2 BL1 BL2] ⇒
    let [g ⊢ BE1] = binp_latetoearly [g ⊢ BL1] in
    let [g ⊢ BE2] = bout_latetoearly [g ⊢ BL2] in
    [g ⊢ efs_close2 BE1 BE2]
;
```

Figure A.5: Proof of the second equivalence theorem.

Appendix B

Proof of Lemma 2.6

Lemma 2.6. *If $P \equiv Q$ and $P \xrightarrow{\alpha} P'$, then there exists a process Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \equiv Q'$.*

Proof. Let $P \equiv Q$. As previously explained on page 21, we proceed by induction on the structure of $P \equiv Q$; for each case, we first assume $P \xrightarrow{\alpha} P'$, aiming to obtain a transition of the process Q, and then we assume $Q \xrightarrow{\alpha} Q'$, aiming to obtain a transition of P.

$$\begin{tikzcd}
P \arrow[r, phantom, "\equiv"] \arrow[d, "\alpha"] & Q \arrow[d, dashed, "\alpha"] & P \arrow[r, phantom, "\equiv"] \arrow[d, dashed, "\alpha"] & Q \arrow[d, "\alpha"] \\
P' \arrow[r, phantom, "\equiv"] & Q' & P' \arrow[r, phantom, "\equiv"] & Q'
\end{tikzcd}$$

Figure B.1: Graphical representation of Lemma 2.6 statement.

- Par-Unit: we have $P \mid \mathbf{0} \equiv P$.

  First, let $P \mid \mathbf{0} \xrightarrow{\alpha} P'$. By inversion on this transition, we deduce it was obtained through the application of a S-Par-L rule: all other rules would imply that the process $\mathbf{0}$ is involved in some transition, leading to a contradiction. Thus, the transition can be rewritten as $P \mid \mathbf{0} \xrightarrow{\alpha} Q \mid \mathbf{0}$, and we obtain that $P \xrightarrow{\alpha} Q$. Since $Q \mid \mathbf{0} \equiv Q$, this concludes the argument.

  Conversely, if we assume that $P \xrightarrow{\alpha} P'$, we can derive that $P \mid \mathbf{0} \xrightarrow{\alpha} P' \mid \mathbf{0}$ using the S-Par-L rule, and conclude by observing that $P' \mid \mathbf{0} \equiv P'$.

- Par-Comm: we have $P \mid Q \equiv Q \mid P$.

  First, let $P \mid Q \xrightarrow{\alpha} R$ (T1) and perform inversion on this transition. We have the following subcases:

  - S-Par-L: (T1) can be rewritten as $P \mid Q \xrightarrow{\alpha} R \mid Q$, and we have that $P \xrightarrow{\alpha} R$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset$.
    By S-Par-R we obtain $Q \mid P \xrightarrow{\alpha} Q \mid R$, and since $P \mid Q \equiv Q \mid P$ we achieved our goal.
  - S-Par-R: analogous to the previous case.

- S-Com-L: (T1) can be rewritten as $P \mid Q \xrightarrow{\tau} P' \mid Q'$, with $P \xrightarrow{\bar{x}y} P'$ and $Q \xrightarrow{x(y)} Q'$.
  By S-Com-R we obtain $Q \mid P \xrightarrow{\alpha} Q' \mid P'$, and since $P' \mid Q' \equiv Q' \mid P'$ we achieved our goal.
- S-Com-R: analogous to the previous case.
- S-Close-L: (T1) can be rewritten as $P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')$, with $P \xrightarrow{\bar{x}(z)} P'$, $Q \xrightarrow{x(z)} Q'$ and $z \notin \text{fn}(Q)$.
  By S-Close-R we obtain $Q \mid P \xrightarrow{\alpha} (\nu z)(Q' \mid P')$, and since $(\nu z)(P' \mid Q') \equiv (\nu z)(Q' \mid P')$ we achieved our goal.
- S-Close-R: analogous to the previous case.

Conversely, if we assume that $Q \mid P \xrightarrow{\alpha} R$ we can conclude analogously.

- Par-Assoc: we have $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$.

  First, let $P \mid (Q \mid R) \xrightarrow{\alpha} S$, denoted as (T1). We perform inversion on this transition, obtaining the following subcases:

  - S-Par-L: (T1) can be rewritten as $P \mid (Q \mid R) \xrightarrow{\alpha} P' \mid (Q \mid R)$, and we have that $P \xrightarrow{\alpha} P'$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(Q \mid R) = \emptyset$.
    Since $\mathsf{fn}(Q \mid R) = \mathsf{fn}(Q) \cup \mathsf{fn}(R)$ and disjunction is distributive over union, we obtain that both $\mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(R) = \emptyset$. Thus, we can apply a first S-Par-L rule to obtain that $P \mid Q \xrightarrow{\alpha} P' \mid Q$, and we can apply a second S-Par-L rule to obtain that $(P \mid Q) \mid R \xrightarrow{\alpha} (P' \mid Q) \mid R$. The latter process is congruent to $P' \mid (Q \mid R)$ via Par-Assoc, hence achieving our goal.
  - S-Par-R: (T1) can be rewritten as $P \mid (Q \mid R) \xrightarrow{\alpha} P \mid S'$, and we have that $(Q \mid R) \xrightarrow{\alpha} S'$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(P) = \emptyset$. We perform inversion on the transition $(Q \mid R) \xrightarrow{\alpha} S'$, obtaining different subcases.
    * S-Par-L: (T1) can be rewritten as $P \mid (Q \mid R) \xrightarrow{\alpha} P \mid (Q' \mid R)$, and we have that $Q \xrightarrow{\alpha} Q'$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(R) = \emptyset$.
      We can apply a S-Par-R rule to obtain $P \mid Q \xrightarrow{\alpha} P \mid Q'$; then, we apply a S-Par-L rule to obtain $(P \mid Q) \mid R \xrightarrow{\alpha} (P \mid Q') \mid R$. The latter process is congruent to $P \mid (Q' \mid R)$ via Par-Assoc, hence achieving our goal.
    * S-Par-R: (T1) can be rewritten as $P \mid (Q \mid R) \xrightarrow{\alpha} P \mid (Q \mid R')$, and we have that $R \xrightarrow{\alpha} R'$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset$.
      Since $\mathsf{bn}(\alpha) \cap \mathsf{fn}(P \mid Q) = \emptyset$, we can apply a single S-Par-R rule and obtain that $(P \mid Q) \mid R \xrightarrow{\alpha} (P \mid Q) \mid R'$. The latter process is congruent to $P \mid (Q \mid R')$ via Par-Assoc, hence achieving our goal.
    * S-Com-L: (T1) can be rewritten as $P \mid (Q \mid R) \xrightarrow{\tau} P \mid (Q' \mid R')$, and we have that $Q \xrightarrow{\bar{x}y} Q'$ and $R \xrightarrow{x(y)} R'$.
      By applying a S-Par-R rule we obtain that $P \mid Q \xrightarrow{\bar{x}y} P \mid Q'$; then we apply a S-Com-L rule and obtain that $(P \mid Q) \mid R \xrightarrow{\tau} (P \mid Q') \mid R'$. The latter process is congruent to $P \mid (Q' \mid R')$ via Par-Assoc, hence achieving our goal.
    * S-Com-R: analogous to the previous case.

- $*$ S-Close-L: previously proved on page 22.
- $*$ S-Close-R: (T1) is rewritten as $P \mid (Q \mid R) \xrightarrow{\tau} P \mid ((\nu z)(Q' \mid R'))$, and we have that $z \notin \mathsf{fn}(Q)$, $Q \xrightarrow{x(z)} Q'$ and $R \xrightarrow{\bar{x}(z)} R'$.
  Through a S-Par-R rule we obtain that $P \mid Q \xrightarrow{x(z)} P \mid Q'$. Then, since $R \xrightarrow{\bar{x}(z)} R'$, through Lemma 2.3 we obtain that $z \in \mathsf{bn}(R)$; for the variable convention we can assume that the bound name z is not free in $P \mid Q$. Thus, through a S-Close-R rule we obtain that $(P \mid Q) \mid R \xrightarrow{\tau} (\nu z)((P \mid Q') \mid R')$. Finally, we conclude analogously to the previous case.

– S-Com-L: (T1) can be rewritten as $P \mid (Q \mid R) \xrightarrow{\tau} P' \mid S'$, and we have that $P \xrightarrow{\bar{x}y} P'$ and $Q \mid R \xrightarrow{x(y)} S'$. We perform inversion on the transition $Q \mid R \xrightarrow{x(y)} S'$ obtaining two subcases:
- $*$ S-Par-L: (T1) can be rewritten as $P \mid (Q \mid R) \xrightarrow{\tau} P' \mid (Q' \mid R)$, and we have that $Q \xrightarrow{x(y)} Q'$.
  By S-Com-L we obtain that $P \mid Q \xrightarrow{\tau} P' \mid Q'$; by S-Par-L we obtain that $(P \mid Q) \mid R \xrightarrow{\tau} (P' \mid Q') \mid R$. The latter process is congruent to the process $P' \mid (Q' \mid R)$ via Par-Assoc, hence achieving our goal.
- $*$ S-Par-R: (T1) can be rewritten as $P \mid (Q \mid R) \xrightarrow{\tau} P' \mid (Q \mid R')$, and we have that $R \xrightarrow{x(y)} R'$.
  By S-Par-L we obtain that $P \mid Q \xrightarrow{\bar{x}y} P' \mid Q$; by S-Com-L we obtain that $(P \mid Q) \mid R \xrightarrow{\tau} (P' \mid Q) \mid R'$. The latter process is congruent to the process $P' \mid (Q \mid R')$ via Par-Assoc, hence achieving our goal.

– S-Com-R: (T1) can be rewritten as $P \mid (Q \mid R) \xrightarrow{\tau} P' \mid S'$, and we have that $P \xrightarrow{x(y)} P'$ and $Q \mid R \xrightarrow{\bar{x}y} S'$.
As in the previous case, by inversion we have that the transition
$Q \mid R \xrightarrow{x(y)} S'$ can be obtained either through S-Par-L or S-Par-R; by replacing S-Com-L with S-Com-R in the former argument, we come to the conclusion.

– S-Close-L: (T1) can be rewritten as $P \mid (Q \mid R) \xrightarrow{\tau} (\nu z)(P' \mid S')$, and we have that $P \xrightarrow{\bar{x}(z)} P'$, $Q \mid R \xrightarrow{x(z)} S'$ and $z \notin \mathsf{fn}(Q \mid R)$.
Since $z \notin \mathsf{fn}(Q \mid R)$ we obtain that both $z \notin \mathsf{fn}(Q)$ and $z \notin \mathsf{fn}(R)$. We perform inversion on the transition $Q \mid R \xrightarrow{x(z)} S'$ obtaining two subcases:
- $*$ S-Par-L: (T1) can be rewritten as $P \mid (Q \mid R) \xrightarrow{\tau} (\nu z)(P' \mid (Q' \mid R))$, and we have that $Q \xrightarrow{x(z)} Q'$.
  Since $z \notin \mathsf{fn}(Q)$ we can apply the S-Close-L rule and obtain that $P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')$; by S-Par-L we obtain that
  $(P \mid Q) \mid R \xrightarrow{\tau} ((\nu z)(P' \mid Q')) \mid R$.
  As in the former case S-Close-L proved on page 22, we obtain that the latter process is congruent to the process $(\nu z)(P' \mid (Q' \mid R))$ by applying a combination of Sc-Ext-Par, C-Res and monoid axioms for parallel composition.
- $*$ S-Par-R: (T1) can be rewritten as $P \mid (Q \mid R) \xrightarrow{\tau} (\nu z)(P' \mid (Q \mid R'))$, and we have that $R \xrightarrow{x(z)} R'$.

Since $z \notin \mathsf{fn}(Q)$ we can apply the S-Par-L rule and obtain that $P \mid Q \xrightarrow{\bar{x}(z)} P' \mid Q$; since $z \notin \mathsf{fn}(R)$ we can apply the S-Close-L rule and obtain that $(P \mid Q) \mid R \xrightarrow{\tau} (\nu z)((P' \mid Q) \mid R')$.
We conclude by observing that the latter process is congruent to the process $(\nu z)(P' \mid (Q \mid R'))$ through the application of a Par-Assoc rule followed by a C-Res rule.

- S-Close-R: (T1) can be rewritten as $P \mid (Q \mid R) \xrightarrow{\tau} (\nu z)(P' \mid S')$, and we have that $P \xrightarrow{x(z)} P'$, $Q \mid R \xrightarrow{\bar{x}(z)} S'$ and $z \notin \mathsf{fn}(P)$. We perform inversion on the transition $Q \mid R \xrightarrow{\bar{x}(z)} S'$ obtaining two subcases:
  - S-Par-L: (T1) can be rewritten as $P \mid (Q \mid R) \xrightarrow{\tau} (\nu z)(P' \mid (Q' \mid R))$, and we have that $Q \xrightarrow{\bar{x}(z)} Q'$ and $z \notin \mathsf{fn}(R)$.
  Through S-Close-R we obtain that $P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')$. Through S-Par-L we obtain that $(P \mid Q) \mid R \xrightarrow{\tau} (\nu z)(P' \mid Q') \mid R$.
  As in the former case S-Close-L proved on page 22, since $z \notin \mathsf{fn}(R)$ we obtain that the process $(\nu z)(P' \mid Q') \mid R$ is congruent to the process $(\nu z)(P' \mid (Q' \mid R))$ through a combination of Sc-Ext-Par, C-Res and monoid rules for parallel composition.
  - S-Par-R: (T1) can be rewritten as $P \mid (Q \mid R) \xrightarrow{\tau} (\nu z)(P' \mid (Q \mid R'))$, and we have that $R \xrightarrow{\bar{x}(z)} R'$ and $z \notin \mathsf{fn}(Q)$.
  Through S-Par-L we obtain that $P \mid Q \xrightarrow{x(z)} P' \mid Q$.
  Since $z \notin \mathsf{fn}(P)$ and $z \notin \mathsf{fn}(Q)$, through S-Close-R we obtain that $(P \mid Q) \mid R \xrightarrow{\tau} (\nu z)((P' \mid Q) \mid R')$. The latter process is congruent to the process $(\nu z)(P' \mid (Q \mid R'))$ through the application of a Par-Assoc rule followed by a C-Res rule.

Conversely, if we assume that $(P \mid Q) \mid R \xrightarrow{\alpha} S$ we can conclude analogously.

- Sc-Ext-Zero: we have $(\nu x)\,\mathbf{0} \equiv \mathbf{0}$. Since both processes do not undergo any transition, there is nothing to prove.

- Sc-Ext-Par: we have $(\nu x)P \mid Q \equiv (\nu x)(P \mid Q)$, with $x \notin \mathsf{fn}(Q)$.
First, let $(\nu x)P \mid Q \xrightarrow{\alpha} R$, denoted as (T1). We perform inversion on this transition, obtaining the following subcases.
  - S-Par-L: (T1) can be rewritten as $(\nu x)P \mid Q \xrightarrow{\alpha} S \mid Q$, and we have that $(\nu x)P \xrightarrow{\alpha} S$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset$. By inversion on the transition $(\nu x)P \xrightarrow{\alpha} S$ we obtain two subcases:
    - S-Res: (T1) can be rewritten as $(\nu x)P \mid Q \xrightarrow{\alpha} (\nu x)P' \mid Q$, and we have that $x \notin \mathsf{n}(\alpha)$ and $P \xrightarrow{\alpha} P'$.
    Since $\mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset$ we can apply a S-Par-L rule and obtain that $P \mid Q \xrightarrow{\alpha} P' \mid Q$. Since $x \notin \mathsf{n}(\alpha)$ we can apply a S-Res rule and obtain that $(\nu x)(P \mid Q) \xrightarrow{\alpha} (\nu x)(P' \mid Q)$. Finally we observe that, since $x \notin \mathsf{fn}(Q)$, we can say that the latter process is congruent to the process $(\nu x)P' \mid Q$ via Sc-Ext-Par, hence achieving our goal.
    - S-Open: (T1) can be rewritten as $(\nu x)P \mid Q \xrightarrow{\bar{y}(x)} S \mid Q$, and we have that $P \xrightarrow{\bar{y}x} S$ and $x \neq y$.

Through S-Par-L we get that $P \mid Q \xrightarrow{\bar{y}x} S \mid Q$; since $x \neq y$ we can apply a S-Open rule and obtain that $(\nu x)(P \mid Q) \xrightarrow{\bar{y}(x)} S \mid Q$, hence achieving our goal.

– S-Par-R: previously proved on page 22.

– S-Com-L: we have $\alpha = \tau$, $(\nu x)P \xrightarrow{\bar{z}w} S$ and $Q \xrightarrow{z(w)} Q'$. By inversion on the first of these transitions through a S-Res rule we deduce that (T1) can be rewritten as $(\nu x)P \mid Q \xrightarrow{\tau} (\nu x)P' \mid Q'$ and we obtain that $x \neq z, w$ and $P \xrightarrow{\bar{z}w} P'$.

  Through S-Com-L we obtain that $P \mid Q \xrightarrow{\tau} P' \mid Q'$, and through S-Res we obtain that $(\nu x)(P \mid Q) \xrightarrow{\tau} (\nu x)(P' \mid Q')$. Since $Q \xrightarrow{z(w)} Q'$, $x \notin \text{fn}(Q)$ and $x \neq z, w$, through Lemma 2.4 we obtain that $x \notin \text{fn}(Q')$; hence, we conclude by observing that $(\nu x)P' \mid Q'$ is congruent to $(\nu x)(P' \mid Q')$ through a Sc-Ext-Par rule.

– S-Com-R: analogous to the previous case.

– S-Close-L: (T1) can be rewritten as $(\nu x)P \mid Q \xrightarrow{\tau} (\nu w)(S \mid Q')$, and we have that $(\nu x)P \xrightarrow{\bar{z}(w)} S$, $Q \xrightarrow{z(w)} Q'$ and $w \notin \mathsf{fn}(Q)$. By inversion on the transition $(\nu x)P \xrightarrow{\bar{z}(w)} S$ we have two subcases.

  ∗ S-Res: (T1) can be rewritten as $(\nu x)P \mid Q \xrightarrow{\tau} (\nu w)((\nu x)P' \mid Q')$, and we have that $x \neq z, w$ and $P \xrightarrow{\bar{z}(w)} P'$.
  Through S-Close-L we obtain that $P \mid Q \xrightarrow{\tau} (\nu w)(P' \mid Q')$ and through S-Res we have that $(\nu x)(P \mid Q) \xrightarrow{\tau} (\nu x)((\nu w)(P' \mid Q'))$. Since $Q \xrightarrow{z(w)} Q'$, $x \notin \mathsf{fn}(Q)$ and $x \neq z, w$, through Lemma 2.4 we obtain that $x \notin \text{fn}(Q')$; thus, we can apply a Sc-Ext-Par rule to deduce that $(\nu x)P' \mid Q'$ is congruent to $(\nu x)(P' \mid Q')$.
  Through a C-Res rule we obtain that $(\nu w)((\nu x)P' \mid Q')$ is congruent to $(\nu w)((\nu x)(P' \mid Q'))$; the latter process is congruent to $(\nu x)((\nu w)(P' \mid Q'))$ via Sc-Ext-Res, hence achieving our goal.

  ∗ S-Open: (T1) can be rewritten as $(\nu x)P \mid Q \xrightarrow{\tau} (\nu x)(P' \mid Q')$, and we have that $z \neq x$ and $P \xrightarrow{\bar{z}x} P'$.
  First, we apply a S-Com-L rule to obtain that $P \mid Q \xrightarrow{\tau} P' \mid Q'$; then, through S-Res we obtain that $(\nu x)(P \mid Q) \xrightarrow{\tau} (\nu x)(P' \mid Q')$. Since there are no congruences to show, we reached the conclusion.

– S-Close-R: we have $\alpha = \tau$, $(\nu x)P \xrightarrow{z(w)} S$, $Q \xrightarrow{\bar{z}(w)} Q'$ and $w \notin \mathsf{fn}((\nu x)P)$. By inversion on the transition $(\nu x)P \xrightarrow{z(w)} S$ through a S-Res rule we obtain that (T1) is rewritten as $(\nu x)P \mid Q \xrightarrow{\tau} (\nu w)((\nu x)P' \mid Q')$, and we have that $x \neq z, w$ and $P \xrightarrow{z(w)} P'$.

  Since $w \notin \mathsf{fn}((\nu x)P)$ and $x \neq w$ we have that $w \notin \mathsf{fn}(P)$; so we can apply a S-Close-R rule and obtain that $P \mid Q \xrightarrow{\tau} (\nu w)(P' \mid Q')$. Finally we conclude analogously to the S-Res subcase of the previous case (first applying S-Res, then applying the same congruence rules).

Conversely, let $(\nu x)(P \mid Q) \xrightarrow{\alpha} R$, denoted as (T2). By inversion on this transition we obtain two subcases:

- S-Res: we have that $x \notin \mathsf{n}(\alpha)$ and $P \mid Q \xrightarrow{\alpha} S$. We perform inversion on this transition, obtaining the following subcases.
    - S-Par-L: (T2) can be rewritten as $(\nu x)(P \mid Q) \xrightarrow{\alpha} (\nu x)(P' \mid Q)$, and we have that $P \xrightarrow{\alpha} P'$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset$.
      Through S-Res we obtain that $(\nu x)P \xrightarrow{\alpha} (\nu x)P'$; through S-Par-L we obtain that $(\nu x)P \mid Q \xrightarrow{\alpha} (\nu x)P' \mid Q$. The latter process is congruent to the process $(\nu x)(P' \mid Q)$ through a Sc-Ext-Par rule.
    - S-Par-R: (T2) can be rewritten as $(\nu x)(P \mid Q) \xrightarrow{\alpha} (\nu x)(P \mid Q')$, and we have that $Q \xrightarrow{\alpha} Q'$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(P) = \emptyset$.
      Since $\mathsf{fn}((\nu x)P) \subseteq \mathsf{fn}(P)$, we have that $\mathsf{bn}(\alpha) \cap \mathsf{fn}((\nu x)P) = \emptyset$; thus, through a S-Par-R rule we obtain that $(\nu x)P \mid Q \xrightarrow{\alpha} (\nu x)P \mid Q'$. Since $Q \xrightarrow{\alpha} Q'$, $x \notin \mathsf{fn}(Q)$ and $x \notin \mathsf{n}(\alpha)$, through Lemma 2.4 we obtain that $x \notin \mathsf{fn}(Q')$; hence, we obtain that the process $(\nu x)P \mid Q'$ is congruent to the process $(\nu x)(P \mid Q')$ through a Sc-Ext-Par rule.
    - S-Com-L: previously proved on page 23.
    - S-Com-R: (T2) can be rewritten as $(\nu x)(P \mid Q) \xrightarrow{\tau} (\nu x)(P' \mid Q')$, and we have that $P \xrightarrow{z(w)} P'$ and $Q \xrightarrow{\bar{z}w} Q'$.
      Since $Q \xrightarrow{\bar{z}w} Q'$, by applying Lemma 2.1 we have that both z and w belong to $\mathsf{fn}(Q)$; however, by hypothesis $x \notin \mathsf{fn}(Q)$, thus we conclude that $x \neq z, w$. As a result, we can apply the S-Res rule to obtain the transition $(\nu x)P \xrightarrow{z(w)} (\nu x)P'$. Then, through a S-Com-R we obtain that $(\nu x)P \mid Q \xrightarrow{\tau} (\nu x)P' \mid Q'$.
      Since $Q \xrightarrow{\bar{z}w} Q'$, $x \notin \mathsf{fn}(Q)$ and $x \neq z, w$, through Lemma 2.4 we deduce that $x \notin \mathsf{fn}(Q')$; hence, through a Sc-Ext-Par rule we obtain that the process $(\nu x)P' \mid Q'$ is congruent to the process $(\nu x)(P' \mid Q')$.
    - S-Close-L: previously proved on page 23.
    - S-Close-R: (T2) is rewritten as $(\nu x)(P \mid Q) \xrightarrow{\tau} (\nu x)((\nu w)(P' \mid Q'))$, and we have that $P \xrightarrow{z(w)} P'$, $Q \xrightarrow{\bar{z}(w)} Q'$ and $w \notin \mathsf{fn}(P)$.
      Since $Q \xrightarrow{\bar{z}(w)} Q'$, by applying Lemma 2.3 we have that $z \in \mathsf{fn}(Q)$ and $w \in \mathsf{bn}(Q)$. By hypothesis $x \notin \mathsf{fn}(Q)$, hence $x \neq z$. Analogously to the previous case, since w is bound in Q and the restriction (νx) in the process $((\nu x)(P \mid Q))$ is outermost, we can assume that x and w are different up to α-renaming. For this reason, we can proceed as in the previous case, replacing S-Close-L with S-Close-R in the former argument.
- S-Open: previously proved on page 23.

• Sc-Ext-Res: we have $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$.

First, let $(\nu x)(\nu y)P \xrightarrow{\alpha} Q$, denoted as (T1). We perform inversion on this transition, obtaining the following subcases.

- S-Res: we have $(\nu y)P \xrightarrow{\alpha} Q'$ and $x \notin \mathsf{n}(\alpha)$. By inversion on this transition we obtain two subcases:
    - S-Res: (T1) can be rewritten as $(\nu x)(\nu y)P \xrightarrow{\alpha} (\nu x)(\nu y)P'$, and we have $P \xrightarrow{\alpha} P'$ and $y \notin \mathsf{n}(\alpha)$.

Through S-Res we obtain that $(\nu x)P \xrightarrow{\alpha} (\nu x)P'$; through S-Res we obtain that $(\nu y)(\nu x)P \xrightarrow{\alpha} (\nu y)(\nu x)P'$. The latter process is congruent to the process $(\nu x)(\nu y)P'$ through Sc-Ext-Res.

  * S-Open: (T1) can be rewritten as $(\nu x)(\nu y)P \xrightarrow{\bar{z}(y)} (\nu x)P'$, and we have $P \xrightarrow{\bar{z}y} P'$ and $y \neq z$; since $x \notin \mathsf{n}(\alpha)$, we also have $x \neq y, z$.

    Through S-Res we obtain that $(\nu x)P \xrightarrow{\bar{z}y} (\nu x)P'$; through S-Open we obtain that $(\nu y)(\nu x)P \xrightarrow{\bar{z}(y)} (\nu x)P'$, hence achieving our goal.

- S-Open: we have $\alpha = \bar{z}(x)$, $(\nu y)P \xrightarrow{\bar{z}x} Q$ and $x \neq z$. By inversion on this transition through S-Res we obtain that $y \neq x, z$ and $P \xrightarrow{\bar{z}x} P'$. (T1) can be rewritten as $(\nu x)(\nu y)P \xrightarrow{\bar{z}(x)} (\nu y)P'$.

  Through S-Open we obtain that $(\nu x)P \xrightarrow{\bar{z}(x)} P'$; through S-Res we obtain that $(\nu y)(\nu x)P \xrightarrow{\bar{z}(x)} (\nu y)P'$, hence achieving our goal.

Conversely, if we assume that $(\nu y)(\nu x)P \xrightarrow{\alpha} Q$ we can conclude analogously.

- C-In: we have $x(z).P \equiv x(z).Q$, given that $P \equiv Q$.

  First, let $x(z).P \xrightarrow{\alpha} P'$. By inversion through S-In we obtain that $\alpha = x(y)$ and $P' = P\{y/z\}$. Through S-In we obtain the transition $x(z).Q \xrightarrow{x(y)} Q\{y/z\}$; since $P \equiv Q$, we also have that $P\{y/z\} \equiv Q\{y/z\}$, hence achieving our goal.

  Conversely, if we assume that $x(z).Q \xrightarrow{\alpha} Q'$ we can conclude analogously.

- C-Out: we have $\bar{x}y.P \equiv \bar{x}y.Q$, given that $P \equiv Q$.

  First, let $\bar{x}y.P \xrightarrow{\alpha} P'$. By inversion through S-Out we obtain that $\alpha = \bar{x}y$ and $P' = P$. Through S-Out we obtain that $\bar{x}y.Q \xrightarrow{\bar{x}y} Q$; since $P \equiv Q$ by hypothesis, we reached the conclusion.

  Conversely, if we assume that $\bar{x}y.Q \xrightarrow{\alpha} Q'$ we can conclude analogously.

- C-Par: we have $P \mid Q \equiv P' \mid Q$, given that $P \equiv P'$.

  First, let $P \mid Q \xrightarrow{\alpha} R$, denoted as (T1). We perform inversion on this transition, obtaining the following subcases.

  - S-Par-L: (T1) can be rewritten as $P \mid Q \xrightarrow{\alpha} S \mid Q$, and we have $P \xrightarrow{\alpha} S$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset$.

    By applying the inductive hypothesis on the set $\{P \equiv P', P \xrightarrow{\alpha} S\}$ we obtain that $P' \xrightarrow{\alpha} S'$ and $S \equiv S'$. Through S-Par-L we obtain that $P' \mid Q \xrightarrow{\alpha} S' \mid Q$. We conclude by observing that the process $S \mid Q$ is congruent to the process $S' \mid Q$ through C-Par.

  - S-Par-R: previously proved on page 24.
  - S-Com-L: (T1) can be rewritten as $P \mid Q \xrightarrow{\tau} S \mid Q'$, and we have that $P \xrightarrow{\bar{x}y} S$ and $Q \xrightarrow{x(y)} Q'$.

    By applying the inductive hypothesis on the set $\{P \equiv P', P \xrightarrow{\bar{x}y} S\}$ we obtain that $P' \xrightarrow{\bar{x}y} S'$ and $S \equiv S'$. Through S-Com-L we obtain that $P' \mid Q \xrightarrow{\tau} S' \mid Q'$. The process $S \mid Q'$ is congruent to the process $S' \mid Q'$ through C-Par, hence achieving our goal.

- S-Com-R: analogous to the previous case.
- S-Close-L: (T1) can be rewritten as $P \mid Q \xrightarrow{\tau} (\nu z)(S \mid Q')$, and we have that $P \xrightarrow{\bar{x}(z)} S$, $Q \xrightarrow{x(z)} Q'$ and $z \notin \mathsf{fn}(Q)$.

  By applying the inductive hypothesis on the set $\{P \equiv P', P \xrightarrow{\bar{x}(z)} S\}$ we obtain that $P' \xrightarrow{\bar{x}(z)} S'$ and $S \equiv S'$. Through S-Close-L we obtain that $P' \mid Q \xrightarrow{\tau} (\nu z)(S' \mid Q')$.
  Since $S \equiv S'$, through C-Par we obtain that $S \mid Q' \equiv S' \mid Q'$; finally, through C-Res we obtain that $(\nu z)(S \mid Q') \equiv (\nu z)(S' \mid Q')$, thus reaching the conclusion.
- S-Close-R: (T1) can be rewritten as $P \mid Q \xrightarrow{\tau} (\nu z)(S \mid Q')$, and we have that $P \xrightarrow{x(z)} S$, $Q \xrightarrow{\bar{x}(z)} Q'$ and $z \notin \mathsf{fn}(P)$.

  By applying the inductive hypothesis on the set $\{P \equiv P', P \xrightarrow{x(z)} S\}$ we obtain that $P' \xrightarrow{x(z)} S'$ and $S \equiv S'$. Since $P \equiv P'$ and $z \notin \mathsf{fn}(P)$, through Lemma 2.5 we obtain that $z \notin \mathsf{fn}(P')$. Thus, through S-Close-R we obtain that $P' \mid Q \xrightarrow{\tau} (\nu z)(S' \mid Q')$.
  Since $S \equiv S'$, through C-Par we obtain that $S \mid Q' \equiv S' \mid Q'$; finally, through C-Res we obtain that $(\nu z)(S \mid Q') \equiv (\nu z)(S' \mid Q')$, thus reaching the conclusion.

Conversely, if we assume that $P' \mid Q \xrightarrow{\alpha} R$ we can conclude analogously.

- C-Res: we have $(\nu x)P \equiv (\nu x)Q$, given that $P \equiv Q$.

  First, let $(\nu x)P \xrightarrow{\alpha} R$, denoted as (T1). We perform inversion on this transition, obtaining the following subcases.

  - S-Res: (T1) can be rewritten as $(\nu x)P \xrightarrow{\alpha} (\nu x)P'$, and we have $P \xrightarrow{\alpha} P'$ and $x \notin \mathsf{n}(\alpha)$.
    By applying the inductive hypothesis on the set $\{P \equiv Q, P \xrightarrow{\alpha} P'\}$ we obtain that $Q \xrightarrow{\alpha} Q'$ and $P' \equiv Q'$. Through S-Res we obtain that $(\nu x)Q \xrightarrow{\alpha} (\nu x)Q'$. Since $P' \equiv Q'$, the process $(\nu x)Q'$ is congruent to the process $(\nu x)P'$ through C-Res.
  - S-Open: (T1) can be rewritten as $(\nu x)P \xrightarrow{\bar{z}(x)} R$, and we have $P \xrightarrow{\bar{z}x} R$ and $z \neq x$.
    By applying the inductive hypothesis on the set $\{P \equiv Q, P \xrightarrow{\bar{z}x} R\}$ we obtain that $Q \xrightarrow{\bar{z}x} S$ and $R \equiv S$. Through S-Open we obtain that $(\nu x)Q \xrightarrow{\bar{z}(x)} S$, and since $R \equiv S$ we achieved our goal.

  Conversely, if we assume that $(\nu x)Q \xrightarrow{\alpha} S$ we can conclude analogously.

- C-Ref: we have $P \equiv P$; the proof is immediate.
- C-Sym: the conclusion holds automatically because of the initial remark.
- C-Trans: we have $P \equiv R$, given that $P \equiv Q$ and $Q \equiv R$.

  First, assume $P \xrightarrow{\alpha} P'$. By applying the inductive hypothesis on the set $\{P \equiv Q, P \xrightarrow{\alpha} P'\}$ we obtain that $Q \xrightarrow{\alpha} Q'$ and $P' \equiv Q'$. By applying the inductive hypothesis on the set $\{Q \equiv R, Q \xrightarrow{\alpha} Q'\}$ we obtain that $R \xrightarrow{\alpha} R'$ and $Q' \equiv R'$. Through C-Trans we obtain that $P' \xrightarrow{\alpha} R'$, hence reaching the conclusion.

Conversely, if we assume that $R \xrightarrow{\alpha} R'$ we can conclude analogously.

□

Appendix C

Formalization of Lemma 2.6

Proof of the Strengthening Lemma

```
rec strengthen_fstep: (g:ctx) {F:[g,x:names ⊢ fstep P[..] A Q]}
  → ex_str_fstep [g,x:names ⊢ F] =
/ total f (strengthen_fstep _ _ _ _ f) /
mlam F ⇒ case [_,x:names ⊢ F] of
  | [g,x:names ⊢ fs_out] ⇒ ex_fstep [g,x:names ⊢ F] [g ⊢ fs_out]
        [g,x:names ⊢ frefl] [g,x:names ⊢ prefl]
  | [g,x:names ⊢ fs_par1 F1[..,x]] ⇒
    let ex_fstep [g,x:names ⊢ F1[..,x]] [g ⊢ F1'] e1 e2
        = strengthen_fstep [g,x:names ⊢ F1[..,x]] in
    let [g,x:names ⊢ frefl] = e1 in
    let [g,x:names ⊢ prefl] = e2 in
        ex_fstep [g,x:names ⊢ fs_par1 F1[..,x]] [g ⊢ fs_par1 F1']
        [g,x:names ⊢ frefl] [g,x:names ⊢ prefl]
  | [g,x:names ⊢ fs_par2 F2[..,x]] ⇒
    let ex_fstep [g,x:names ⊢ F2[..,x]] [g ⊢ F2'] e1 e2
        = strengthen_fstep [g,x:names ⊢ F2[..,x]] in
    let [g,x:names ⊢ frefl] = e1 in
    let [g,x:names ⊢ prefl] = e2 in
        ex_fstep [g,x:names ⊢ fs_par2 F2[..,x]] [g ⊢ fs_par2 F2']
        [g,x:names ⊢ frefl] [g,x:names ⊢ prefl]
  | [g,x:names ⊢ fs_com1 F1[..,x] B1[..,x]] ⇒
    let ex_fstep [g,x:names ⊢ F1[..,x]] [g ⊢ F1'] e1 e2
        = strengthen_fstep [g,x:names ⊢ F1[..,x]] in
    let [g,x:names ⊢ frefl] = e1 in
    let [g,x:names ⊢ prefl] = e2 in
    let ex_bstep [g,x:names ⊢ B1[..,x]] [g ⊢ B1'] e1' e2'
        = strengthen_bstep [g,x:names ⊢ B1[..,x]] in
    let [g,x:names ⊢ brefl] = e1' in
    let [g,x:names,z:names ⊢ prefl] = e2' in
        ex_fstep [g,x:names ⊢ fs_com1 F1[..,x] B1[..,x]]
        [g ⊢ fs_com1 F1' B1'] [g,x:names ⊢ frefl] [g,x:names ⊢ prefl]
  | [g,x:names ⊢ fs_com2 B1[..,x] F1[..,x]] ⇒
    let ex_fstep [g,x:names ⊢ F1[..,x]] [g ⊢ F1'] e1 e2
        = strengthen_fstep [g,x:names ⊢ F1[..,x]] in
    let [g,x:names ⊢ frefl] = e1 in
    let [g,x:names ⊢ prefl] = e2 in
    let ex_bstep [g,x:names ⊢ B1[..,x]] [g ⊢ B1’] e1’ e2’
        = strengthen_bstep [g,x:names ⊢ B1[..,x]] in
    let [g,x:names ⊢ brefl] = e1’ in
    let [g,x:names,z:names ⊢ prefl] = e2’ in
        ex_fstep [g,x:names ⊢ fs_com2 B1[..,x] F1[..,x]]
        [g ⊢ fs_com2 B1’ F1’] [g,x:names ⊢ frefl] [g,x:names ⊢ prefl]
  | [g,x:names ⊢ fs_res \y.F1[..,y,x]] ⇒
    let ex_fstep [g,y:names,x:names ⊢ F1[..,y,x]] [g,y:names ⊢ F1’[..,y]]
        e1 e2 = strengthen_fstep [g,y:names,x:names ⊢ F1[..,y,x]] in
    let [g,y:names,x:names ⊢ frefl] = e1 in
    let [g,y:names,x:names ⊢ prefl] = e2 in
        ex_fstep [g,x:names ⊢ fs_res \y.F1[..,y,x]]
        [g ⊢ fs_res \y.F1’[..,y]] [g,x:names ⊢ frefl] [g,x:names ⊢ prefl]
  | [g,x:names ⊢ fs_close1 B1[..,x] B2[..,x]] ⇒
    let ex_bstep [g,x:names ⊢ B1[..,x]] [g ⊢ B1’] e1 e2
        = strengthen_bstep [g,x:names ⊢ B1[..,x]] in
    let [g,x:names ⊢ brefl] = e1 in
    let [g,x:names,z:names ⊢ prefl] = e2 in
    let ex_bstep [g,x:names ⊢ B2[..,x]] [g ⊢ B2’] e1’ e2’
        = strengthen_bstep [g,x:names ⊢ B2[..,x]] in
    let [g,x:names ⊢ brefl] = e1’ in
    let [g,x:names,z:names ⊢ prefl] = e2’ in
        ex_fstep [g,x:names ⊢ fs_close1 B1[..,x] B2[..,x]]
        [g ⊢ fs_close1 B1’ B2’] [g,x:names ⊢ frefl] [g,x:names ⊢ prefl]
  | [g,x:names ⊢ fs_close2 B1[..,x] B2[..,x]] ⇒
    let ex_bstep [g,x:names ⊢ B1[..,x]] [g ⊢ B1’] e1 e2
        = strengthen_bstep [g,x:names ⊢ B1[..,x]] in
    let [g,x:names ⊢ brefl] = e1 in
    let [g,x:names,z:names ⊢ prefl] = e2 in
    let ex_bstep [g,x:names ⊢ B2[..,x]] [g ⊢ B2’] e1’ e2’
        = strengthen_bstep [g,x:names ⊢ B2[..,x]] in
    let [g,x:names ⊢ brefl] = e1’ in
    let [g,x:names,z:names ⊢ prefl] = e2’ in
        ex_fstep [g,x:names ⊢ fs_close2 B1[..,x] B2[..,x]]
        [g ⊢ fs_close2 B1’ B2’] [g,x:names ⊢ frefl] [g,x:names ⊢ prefl]

and rec strengthen_bstep: (g:ctx) {B:[g,x:names ⊢ bstep P[..] A
  \z.Q[..,x,z]]} → ex_str_bstep [g,x:names ⊢ B] =
/ total b (strengthen_bstep _ _ _ _ b) /
mlam B ⇒ case [_,x:names ⊢ B] of
  | [g,x:names ⊢ bs_in] ⇒ ex_bstep [g,x:names ⊢ B] [g ⊢ bs_in]
        [g,x:names ⊢ brefl] [g,x:names,z:names ⊢ prefl]
  | [g,x:names ⊢ bs_par1 B1[..,x]] ⇒
    let ex_bstep [g,x:names ⊢ B1[..,x]] [g ⊢ B1’] e1 e2
        = strengthen_bstep [g,x:names ⊢ B1[..,x]] in
    let [g,x:names ⊢ brefl] = e1 in
    let [g,x:names,z:names ⊢ prefl] = e2 in
        ex_bstep [g,x:names ⊢ bs_par1 B1[..,x]] [g ⊢ bs_par1 B1’]
        [g,x:names ⊢ brefl] [g,x:names,z:names ⊢ prefl]
  | [g,x:names ⊢ bs_par2 B2[..,x]] ⇒
    let ex_bstep [g,x:names ⊢ B2[..,x]] [g ⊢ B2’] e1 e2
```

```
        = strengthen_bstep [g,x:names ⊢ B2[..,x]] in
    let [g,x:names ⊢ brefl] = e1 in
    let [g,x:names,z:names ⊢ prefl] = e2 in
        ex_bstep [g,x:names ⊢ bs_par2 B2[..,x]] [g ⊢ bs_par2 B2']
        [g,x:names ⊢ brefl] [g,x:names,z:names ⊢ prefl]
  | [g,x:names ⊢ bs_res \y.B1[..,y,x]] ⇒
    let ex_bstep [g,y:names,x:names ⊢ B1[..,y,x]] [g,y:names ⊢ B1'[..,y]]
        e1 e2 = strengthen_bstep [g,y:names,x:names ⊢ B1[..,y,x]] in
    let [g,y:names,x:names ⊢ brefl] = e1 in
    let [g,y:names,x:names,z:names ⊢ prefl] = e2 in
        ex_bstep [g,x:names ⊢ bs_res \y.B1[..,y,x]]
        [g ⊢ bs_res \y.B1'[..,y]] [g,x:names ⊢ brefl]
        [g,x:names,z:names ⊢ prefl]
  | [g,x:names ⊢ bs_open \y.F1[..,y,x]] ⇒
    let ex_fstep [g,y:names,x:names ⊢ F1[..,y,x]] [g,y:names ⊢ F1'[..,y]]
        e1 e2 = strengthen_fstep [g,y:names,x:names ⊢ F1[..,y,x]] in
    let [g,y:names,x:names ⊢ frefl] = e1 in
    let [g,y:names,x:names ⊢ prefl] = e2 in
        ex_bstep [g,x:names ⊢ bs_open \y.F1[..,y,x]]
        [g ⊢ bs_open \y.F1'[..,y]] [g,x:names ⊢ brefl]
        [g,x:names,z:names ⊢ prefl]
;
```

Proof of Lemma 2.6

```
rec cong_fstepleft_impl_fstepright: (g:ctx) [g ⊢ P cong Q]
  → [g ⊢ fstep P A P'] → [g ⊢ ex_fstepcong P Q A P'] =
/ total c (cong_fstepleft_impl_fstepright _ _ _ _ _ c _)/
fn c ⇒ fn f ⇒ case c of
  | [g ⊢ par_unit] ⇒ let [g ⊢ fs_par1 F1] = f in [g ⊢ fsc F1 par_unit]
  | [g ⊢ par_comm] ⇒
    (case f of
      | [g ⊢ fs_par1 F1] ⇒ [g ⊢ fsc (fs_par2 F1) par_comm]
      | [g ⊢ fs_par2 F2] ⇒ [g ⊢ fsc (fs_par1 F2) par_comm]
      | [g ⊢ fs_com1 F1 B1] ⇒ [g ⊢ fsc (fs_com2 B1 F1) par_comm]
      | [g ⊢ fs_com2 B1 F1] ⇒ [g ⊢ fsc (fs_com1 F1 B1) par_comm]
      | [g ⊢ fs_close1 B1 B2] ⇒
        [g ⊢ fsc (fs_close2 B2 B1) (c_res \x.par_comm)]
      | [g ⊢ fs_close2 B1 B2] ⇒
        [g ⊢ fsc (fs_close1 B2 B1) (c_res \x.par_comm)]
    )
  | [g ⊢ par_assoc] ⇒
    (case f of
      | [g ⊢ fs_par1 F] ⇒ [g ⊢ fsc (fs_par1 (fs_par1 F)) par_assoc]
      | [g ⊢ fs_par2 F] ⇒
        (case [g ⊢ F] of
          | [g ⊢ fs_par1 F'] ⇒
            [g ⊢ fsc (fs_par1 (fs_par2 F')) par_assoc]
          | [g ⊢ fs_par2 F'] ⇒ [g ⊢ fsc (fs_par2 F') par_assoc]
          | [g ⊢ fs_com1 F' B] ⇒
            [g ⊢ fsc (fs_com1 (fs_par2 F') B) par_assoc]
          | [g ⊢ fs_com2 B F'] ⇒
            [g ⊢ fsc (fs_com2 (bs_par2 B) F') par_assoc]
         %| [g ⊢ fs_close1 B1 B2] ⇒ previously proved in section 3.3.2
          | [g ⊢ fs_close2 B1 B2] ⇒
            [g ⊢ fsc (fs_close2 (bs_par2 B1) B2) (c_trans
            par_comm (c_trans sc_ext_par (c_trans (c_res
            \x.par_comm) (c_res \x.par_assoc))))]
        )
      | [g ⊢ fs_com1 F B] ⇒
        (case [g ⊢ B] of
          | [g ⊢ bs_par1 B'] ⇒
            [g ⊢ fsc (fs_par1 (fs_com1 F B')) par_assoc]
          | [g ⊢ bs_par2 B'] ⇒
            [g ⊢ fsc (fs_com1 (fs_par1 F) B') par_assoc]
        )
      | [g ⊢ fs_com2 B F] ⇒
        (case [g ⊢ F] of
          | [g ⊢ fs_par1 F'] ⇒
            [g ⊢ fsc (fs_par1 (fs_com2 B F')) par_assoc]
          | [g ⊢ fs_par2 F'] ⇒
            [g ⊢ fsc (fs_com2 (bs_par1 B) F') par_assoc]
        )
      | [g ⊢ fs_close1 B1 B2] ⇒
        (case [g ⊢ B2] of
          | [g ⊢ bs_par1 B2'] ⇒
            [g ⊢ fsc (fs_par1 (fs_close1 B1 B2'))
            (c_trans (c_res \x.par_assoc) (c_sym sc_ext_par))]
          | [g ⊢ bs_par2 B2'] ⇒
            [g ⊢ fsc (fs_close1 (bs_par1 B1) B2') (c_res \x.par_assoc)]
        )
      | [g ⊢ fs_close2 B1 B2] ⇒
        (case [g ⊢ B2] of
          | [g ⊢ bs_par1 B2'] ⇒
            [g ⊢ fsc (fs_par1 (fs_close2 B1 B2'))
            (c_trans (c_res \x.par_assoc) (c_sym sc_ext_par))]
          | [g ⊢ bs_par2 B2'] ⇒
            [g ⊢ fsc (fs_close2 (bs_par1 B1) B2') (c_res \x.par_assoc)]
        )
    )
  | [g ⊢ sc_ext_zero] ⇒
    let [g ⊢ fs_res \x.F[..,x]] = f in impossible [g, x:names ⊢ F[..,x]]
  | [g ⊢ sc_ext_par] ⇒
    (case f of
      | [g ⊢ fs_par1 F1] ⇒
        let [g ⊢ fs_res \x.F1'[..,x]] = [g ⊢ F1] in
            [g ⊢ fsc (fs_res \x.(fs_par1 F1'[..,x])) sc_ext_par]
     %| [g ⊢ fs_par2 F2] ⇒ previously proved in section 3.3.2
      | [g ⊢ fs_com1 F1 B1] ⇒
        let [g ⊢ fs_res \x.F1'[..,x]] = [g ⊢ F1] in
            [g ⊢ fsc (fs_res \x.(fs_com1 F1'[..,x] B1[..])) sc_ext_par]
      | [g ⊢ fs_com2 B1 F1] ⇒
```

```
          let [g ⊢ bs_res \x.B1'[..,x]] = [g ⊢ B1] in
              [g ⊢ fsc (fs_res \x.(fs_com2 B1'[..,x] F1[..])) sc_ext_par]
        | [g ⊢ fs_close1 B1 B2] ⇒
          (case [g ⊢ B1] of
             | [g ⊢ bs_res \x.B1'[..,x]] ⇒
              [g ⊢ fsc (fs_res \x.(fs_close1 B1'[..,x] B2[..]))
              (c_trans (c_res \w.sc_ext_par) sc_ext_res)]
             | [g ⊢ bs_open \x.F1[..,x]] ⇒
              [g ⊢ fsc (fs_res \x.(fs_com1 F1[..,x] B2[..])) c_ref]
          )
        | [g ⊢ fs_close2 B1 B2] ⇒
          let [g ⊢ bs_res \x.B1'[..,x]] = [g ⊢ B1] in
              [g ⊢ fsc (fs_res \x.(fs_close2 B1'[..,x] B2[..]))
              (c_trans (c_res \w.sc_ext_par) sc_ext_res)]
    )
  | [g ⊢ sc_ext_res] ⇒
    let [g ⊢ fs_res \x.F[..,x]] = f in
    let [g,x:names ⊢ fs_res \y.F'[..,x,y]] = [g,x:names ⊢ F[..,x]] in
        [g ⊢ fsc (fs_res \y.(fs_res \x.F'[..,x,y])) (sc_ext_res)]
  | [g ⊢ c_in \x.C1[..,x]] ⇒ impossible f
  | [g ⊢ c_out C1] ⇒ let [g ⊢ fs_out] = f in [g ⊢ fsc fs_out C1]
  | [g ⊢ c_par C1] ⇒
    (case f of
        | [g ⊢ fs_par1 F1] ⇒
          let [g ⊢ fsc F2 C2] = cong_fstepleft_impl_fstepright [g ⊢ C1]
              [g ⊢ F1] in [g ⊢ fsc (fs_par1 F2) (c_par C2)]
       %| [g ⊢ fs_par2 F1] ⇒ previously proved in section 3.3.2
        | [g ⊢ fs_com1 F1 B1] ⇒
          let [g ⊢ fsc F2 C2] = cong_fstepleft_impl_fstepright [g ⊢ C1]
              [g ⊢ F1] in [g ⊢ fsc (fs_com1 F2 B1) (c_par C2)]
        | [g ⊢ fs_com2 B1 F1] ⇒
          let [g ⊢ bsc B2 \x.C2[..,x]] = cong_bstepleft_impl_bstepright
              [g ⊢ C1][g ⊢ B1] in [g ⊢ fsc(fs_com2 B2 F1)(c_par C2[..,_])]
        | [g ⊢ fs_close1 B1 B2] ⇒
          let [g ⊢ bsc B1' \x.C2[..,x]] =
              cong_bstepleft_impl_bstepright [g ⊢ C1] [g ⊢ B1] in
              [g ⊢ fsc (fs_close1 B1' B2) (c_res \x.(c_par C2[..,x]))]
        | [g ⊢ fs_close2 B1 B2] ⇒
          let [g ⊢ bsc B1' \x.C2[..,x]] =
              cong_bstepleft_impl_bstepright [g ⊢ C1] [g ⊢ B1] in
              [g ⊢ fsc (fs_close2 B1' B2) (c_res \x.(c_par C2[..,x]))]
    )
  | [g ⊢ c_res \x.C[..,x]] ⇒
    let [g ⊢ fs_res \x.F[..,x]] = f in
    let [g,x:names ⊢ fsc F'[..,x] C'[..,x]] =
        cong_fstepleft_impl_fstepright [g,x:names ⊢ C[..,x]]
        [g,x:names ⊢ F[..,x]] in
        [g ⊢ fsc (fs_res \x.F'[..,x]) (c_res \x.C'[..,x])]
  | [g ⊢ c_ref] ⇒ let [g ⊢ F] = f in [g ⊢ fsc F c_ref]
 %| [g ⊢ c_sym C'] ⇒ previously proved in section 3.3.2
  | [g ⊢ c_trans C1 C2] ⇒
```

```
    let [g ⊢ fsc F1 C1'] = cong_fstepleft_impl_fstepright [g ⊢ C1] f in
    let [g ⊢ fsc F2 C2'] = cong_fstepleft_impl_fstepright [g ⊢ C2]
        [g ⊢ F1] in [g ⊢ fsc F2 (c_trans C1' C2')]

and rec cong_fstepright_impl_fstepleft: (g:ctx) [g ⊢ P cong Q]
  → [g ⊢ fstep Q A Q'] → [g ⊢ ex_fstepcong Q P A Q'] =
/ total c (cong_fstepright_impl_fstepleft _ _ _ _ _ c _)/
fn c ⇒ fn f ⇒ case c of
  | [g ⊢ par_unit] ⇒
    let [g ⊢ F] = f in [g ⊢ fsc (fs_par1 F) (c_sym par_unit)]
  | [g ⊢ par_comm] ⇒
    (case f of
       | [g ⊢ fs_par1 F1] ⇒ [g ⊢ fsc (fs_par2 F1) par_comm]
       | [g ⊢ fs_par2 F2] ⇒ [g ⊢ fsc (fs_par1 F2) par_comm]
       | [g ⊢ fs_com1 F1 B1] ⇒ [g ⊢ fsc (fs_com2 B1 F1) par_comm]
       | [g ⊢ fs_com2 B1 F1] ⇒ [g ⊢ fsc (fs_com1 F1 B1) par_comm]
       | [g ⊢ fs_close1 B1 B2] ⇒
         [g ⊢ fsc (fs_close2 B2 B1) (c_res \x.par_comm)]
       | [g ⊢ fs_close2 B1 B2] ⇒
         [g ⊢ fsc (fs_close1 B2 B1) (c_res \x.par_comm)]
    )
  | [g ⊢ par_assoc] ⇒
    (case f of
       | [g ⊢ fs_par1 F] ⇒
         (case [g ⊢ F] of
            | [g ⊢ fs_par1 F'] ⇒ [g ⊢ fsc (fs_par1 F') (c_sym par_assoc)]
            | [g ⊢ fs_par2 F'] ⇒
              [g ⊢ fsc (fs_par2 (fs_par1 F')) (c_sym par_assoc)]
            | [g ⊢ fs_com1 F' B] ⇒
              [g ⊢ fsc (fs_com1 F' (bs_par1 B)) (c_sym par_assoc)]
            | [g ⊢ fs_com2 B F'] ⇒
              [g ⊢ fsc (fs_com2 B (fs_par1 F')) (c_sym par_assoc)]
            | [g ⊢ fs_close1 B1 B2] ⇒
              [g ⊢ fsc (fs_close1 B1 (bs_par1 B2))
              (c_trans sc_ext_par (c_res \x.(c_sym par_assoc)))]
            | [g ⊢ fs_close2 B1 B2] ⇒
              [g ⊢ fsc (fs_close2 B1 (bs_par1 B2))
              (c_trans sc_ext_par (c_res \x.(c_sym par_assoc)))]
         )
       | [g ⊢ fs_par2 F] ⇒
         [g ⊢ fsc (fs_par2 (fs_par2 F)) (c_sym par_assoc)]
       | [g ⊢ fs_com1 F B] ⇒
         (case [g ⊢ F] of
            | [g ⊢ fs_par1 F'] ⇒
              [g ⊢ fsc (fs_com1 F' (bs_par2 B)) (c_sym par_assoc)]
            | [g ⊢ fs_par2 F'] ⇒
              [g ⊢ fsc (fs_par2 (fs_com1 F' B)) (c_sym par_assoc)]
         )
       | [g ⊢ fs_com2 B F] ⇒
         (case [g ⊢ B] of
            | [g ⊢ bs_par1 B'] ⇒
```

```
              [g ⊢ fsc (fs_com2 B' (fs_par2 F)) (c_sym par_assoc)]
            | [g ⊢ bs_par2 B'] ⇒
              [g ⊢ fsc (fs_par2 (fs_com2 B' F)) (c_sym par_assoc)]
          )
        | [g ⊢ fs_close1 B1 B2] ⇒
          (case [g ⊢ B1] of
            | [g ⊢ bs_par1 B1'] ⇒
              [g ⊢ fsc (fs_close1 B1' (bs_par2 B2))
              (c_res \x.(c_sym par_assoc))]
            | [g ⊢ bs_par2 B1'] ⇒
              [g ⊢ fsc (fs_par2 (fs_close1 B1' B2))
              (c_trans (c_res \x.(c_sym par_assoc)) (c_trans (c_res
              \x.par_comm) (c_trans (c_sym sc_ext_par) par_comm)))]
          )
        | [g ⊢ fs_close2 B1 B2] ⇒
          (case [g ⊢ B1] of
            | [g ⊢ bs_par1 B1'] ⇒
              [g ⊢ fsc (fs_close2 B1' (bs_par2 B2))
              (c_res \x.(c_sym par_assoc))]
            | [g ⊢ bs_par2 B1'] ⇒
              [g ⊢ fsc (fs_par2 (fs_close2 B1' B2))
              (c_trans (c_res \x.(c_sym par_assoc)) (c_trans (c_res
              \x.par_comm) (c_trans (c_sym sc_ext_par) par_comm)))]
          )
      )
    | [g ⊢ sc_ext_zero] ⇒ impossible f
    | [g ⊢ sc_ext_par] ⇒ let [g ⊢ fs_res \x.F[..,x]] = f in
      (case [g, x:names ⊢ F[..,x]] of
        | [g,x:names ⊢ fs_par1 F1[..,x]] ⇒
          [g ⊢ fsc (fs_par1 (fs_res \x.F1[..,x])) (c_sym sc_ext_par)]
        | [g,x:names ⊢ fs_par2 F2[..,x]] ⇒
          let ex_fstep [g,x:names ⊢ F2[..,x]] [g ⊢ F2'] e1 e2 =
              strengthen_fstep [g,x:names ⊢ F2[..,x]] in
          let [g,x:names ⊢ frefl] = e1 in
          let [g,x:names ⊢ prefl] = e2 in
              [g ⊢ fsc (fs_par2 F2') (c_sym sc_ext_par)]
       %| [g,x:names ⊢ fs_com1 F1[..,x] B1[..,x]] ⇒
           %  previously proved in section 3.3.2
        | [g,x:names ⊢ fs_com2 B1[..,x] F1[..,x]] ⇒
          let ex_fstep [g,x:names ⊢ F1[..,x]] [g ⊢ F1'] e1 e2 =
              strengthen_fstep [g,x:names ⊢ F1[..,x]] in
          let [g,x:names ⊢ frefl] = e1 in
          let [g,x:names ⊢ prefl] = e2 in
              [g ⊢ fsc (fs_com2 (bs_res \x.B1[..,x]) F1')
              (c_sym sc_ext_par)]
       %| [g,x:names ⊢ fs_close1 B1[..,x] B2[..,x]] ⇒
       %  previously proved in section 3.3.2
        | [g,x:names ⊢ fs_close2 B1[..,x] B2[..,x]] ⇒
          let ex_bstep [g,x:names ⊢ B2[..,x]] [g ⊢ B2'] e1 e2 =
              strengthen_bstep [g,x:names ⊢ B2[..,x]] in
          let [g,x:names ⊢ brefl] = e1 in
        let [g,x:names,z:names ⊢ prefl] = e2 in
            [g ⊢ fsc (fs_close2 (bs_res \x.B1[..,x]) B2')
            (c_trans sc_ext_res (c_res \w.(c_sym sc_ext_par)))]
    )
  | [g ⊢ sc_ext_res] ⇒
    let [g ⊢ fs_res \x.F[..,x]] = f in
    let [g,x:names ⊢ fs_res \y.F'[..,x,y]] = [g,x:names ⊢ F[..,x]] in
        [g ⊢ fsc (fs_res \y.(fs_res \x.F'[..,x,y])) sc_ext_res]
  | [g ⊢ c_in \x.C1[..,x]] ⇒ impossible f
  | [g ⊢ c_out C1] ⇒ let [g ⊢ fs_out] = f in [g ⊢ fsc fs_out (c_sym C1)]
  | [g ⊢ c_par C1] ⇒
    (case f of
      | [g ⊢ fs_par1 F1] ⇒
        let [g ⊢ fsc F2 C2] = cong_fstepright_impl_fstepleft [g ⊢ C1]
            [g ⊢ F1] in [g ⊢ fsc (fs_par1 F2) (c_par C2)]
      | [g ⊢ fs_par2 F1] ⇒ [g ⊢ fsc (fs_par2 F1) (c_par (c_sym C1))]
      | [g ⊢ fs_com1 F1 B1] ⇒
        let [g ⊢ fsc F2 C2] = cong_fstepright_impl_fstepleft [g ⊢ C1]
            [g ⊢ F1] in [g ⊢ fsc (fs_com1 F2 B1) (c_par C2)]
      | [g ⊢ fs_com2 B1 F1] ⇒
        let [g ⊢ bsc B2 \x.C2[..,x]] = cong_bstepright_impl_bstepleft
            [g ⊢ C1][g ⊢ B1] in [g ⊢ fsc(fs_com2 B2 F1)(c_par C2[..,_])]
      | [g ⊢ fs_close1 B1 B2] ⇒
        let [g ⊢ bsc B1' \x.C2[..,x]] = cong_bstepright_impl_bstepleft
            [g ⊢ C1] [g ⊢ B1] in
            [g ⊢ fsc (fs_close1 B1' B2) (c_res \x.(c_par C2[..,x]))]
      | [g ⊢ fs_close2 B1 B2] ⇒
        let [g ⊢ bsc B1' \x.C2[..,x]] = cong_bstepright_impl_bstepleft
            [g ⊢ C1] [g ⊢ B1] in
            [g ⊢ fsc (fs_close2 B1' B2) (c_res \x.(c_par C2[..,x]))]
    )
  | [g ⊢ c_res \x.C[..,x]] ⇒
    let [g ⊢ fs_res \x.F[..,x]] = f in
    let [g,x:names ⊢ fsc F'[..,x] C'[..,x]] =
        cong_fstepright_impl_fstepleft [g,x:names ⊢ C[..,x]]
        [g,x:names ⊢ F[..,x]] in
        [g ⊢ fsc (fs_res \x.F'[..,x]) (c_res \x.C'[..,x])]
  | [g ⊢ c_ref] ⇒ let [g ⊢ F] = f in [g ⊢ fsc F c_ref]
  | [g ⊢ c_sym C'] ⇒
    let [g ⊢ D] = cong_fstepleft_impl_fstepright [g ⊢ C'] f in [g ⊢ D]
  | [g ⊢ c_trans C1 C2] ⇒
    let [g ⊢ fsc F1 C2'] = cong_fstepright_impl_fstepleft [g ⊢ C2] f in
    let [g ⊢ fsc F2 C1'] = cong_fstepright_impl_fstepleft [g ⊢ C1]
        [g ⊢ F1] in [g ⊢ fsc F2 (c_trans C2' C1')]

and rec cong_bstepleft_impl_bstepright: (g:ctx) [g ⊢ P cong Q]
  → [g ⊢ bstep P A \x.P'[..,x]] → [g ⊢ ex_bstepcong P Q A \x.P'[..,x]] =
/ total c (cong_bstepleft_impl_bstepright _ _ _ _ _ c _)/
fn c ⇒ fn b ⇒ case c of
  | [g ⊢ par_unit] ⇒
    let [g ⊢ bs_par1 B1] = b in [g ⊢ bsc B1 \x.par_unit]
```

```
| [g ⊢ par_comm] ⇒
  (case b of
     | [g ⊢ bs_par1 B1] ⇒ [g ⊢ bsc (bs_par2 B1) \x.par_comm]
     | [g ⊢ bs_par2 B2] ⇒ [g ⊢ bsc (bs_par1 B2) \x.par_comm]
  )
| [g ⊢ par_assoc] ⇒
  (case b of
     | [g ⊢ bs_par1 B] ⇒ [g ⊢ bsc (bs_par1 (bs_par1 B)) \x.par_assoc]
     | [g ⊢ bs_par2 B] ⇒
     (case [g ⊢ B] of
        | [g ⊢ bs_par1 B'] ⇒
          [g ⊢ bsc (bs_par1 (bs_par2 B')) \x.par_assoc]
        | [g ⊢ bs_par2 B'] ⇒ [g ⊢ bsc (bs_par2 B') \x.par_assoc]
     )
  )
| [g ⊢ sc_ext_zero] ⇒
  let [g ⊢ bs_res \x.B[..,x]] = b in impossible [g,x:names ⊢ B[..,x]]
| [g ⊢ sc_ext_par] ⇒
  (case b of
     | [g ⊢ bs_par1 B1] ⇒
       (case [g ⊢ B1] of
          | [g ⊢ bs_res \x.B1'[..,x]] ⇒
            [g ⊢ bsc (bs_res \x.(bs_par1 B1'[..,x])) \x.sc_ext_par]
          | [g ⊢ bs_open \x.F1[..,x]] ⇒
            [g ⊢ bsc (bs_open \x.(fs_par1 F1[..,x])) \x.c_ref]
       )
    %| [g ⊢ bs_par2 B2] ⇒ previously proved in section 3.3.2
  )
| [g ⊢ sc_ext_res] ⇒
  (case b of
     | [g ⊢ bs_res \x.B[..,x]] ⇒
       (case [g,x:names ⊢ B[..,x]] of
          | [g,x:names ⊢ bs_res \y.B'[..,x,y]] ⇒
            [g ⊢ bsc (bs_res \y.(bs_res \x.B'[..,x,y])) \x.sc_ext_res]
          | [g,x:names ⊢ bs_open \y.F[..,x,y]] ⇒
            [g ⊢ bsc (bs_open \y.(fs_res \x.F[..,x,y])) \x.c_ref]
       )
     | [g ⊢ bs_open \x.F[..,x]] ⇒
       let [g,x:names ⊢ fs_res \y.F'[..,x,y]] = [g,x:names ⊢ F[..,x]]
       in [g ⊢ bsc (bs_res \y.(bs_open \x.F'[..,x,y])) \x.c_ref]
  )
| [g ⊢ c_in \x.C1[..,x]] ⇒
  let [g ⊢ bs_in] = b in [g ⊢ bsc bs_in \x.C1[..,x]]
| [g ⊢ c_out C1] ⇒ impossible b
| [g ⊢ c_par C1] ⇒
  (case b of
     | [g ⊢ bs_par1 B1] ⇒
       let [g ⊢ bsc B2 \x.C2[..,x]] = cong_bstepleft_impl_bstepright
       [g ⊢ C1] [g ⊢ B1] in [g ⊢ bsc (bs_par1 B2) \x.(c_par C2[..,x])]
    %| [g ⊢ bs_par2 B1] ⇒ previously proved in section 3.3.2
  )
  | [g ⊢ c_res \x.C[..,x]] ⇒
    (case b of
       | [g ⊢ bs_res \x.B[..,x]] ⇒
         let [g,x:names ⊢ bsc B'[..,x] \y.C'[..,x,y]] =
             cong_bstepleft_impl_bstepright [g,x:names ⊢ C[..,x]]
             [g,x:names ⊢ B[..,x]] in
             [g ⊢ bsc (bs_res \x.B'[..,x]) \y.(c_res \x.C'[..,x,y])]
       | [g ⊢ bs_open \x.F[..,x]] ⇒
         let [g,x:names ⊢ fsc F'[..,x] C'[..,x]] =
             cong_fstepleft_impl_fstepright [g,x:names ⊢ C[..,x]]
             [g,x:names ⊢ F[..,x]] in
             [g ⊢ bsc (bs_open \x.F'[..,x]) \x.C'[..,x]]
    )
  | [g ⊢ c_ref] ⇒ let [g ⊢ B] = b in [g ⊢ bsc B \x.c_ref]
  | [g ⊢ c_sym C'] ⇒
    let [g ⊢ D] = cong_bstepright_impl_bstepleft [g ⊢ C'] b in [g ⊢ D]
  | [g ⊢ c_trans C1 C2] ⇒
    let [g ⊢ bsc B1 \x.C1'[..,x]] =
        cong_bstepleft_impl_bstepright [g ⊢ C1] b in
    let [g ⊢ bsc B2 \x.C2'[..,x]] =
        cong_bstepleft_impl_bstepright [g ⊢ C2] [g ⊢ B1] in
        [g ⊢ bsc B2 \x.(c_trans C1'[..,x] C2'[..,x])]


and rec cong_bstepright_impl_bstepleft: (g:ctx) [g ⊢ P cong Q]
  → [g ⊢ bstep Q A \x.Q'[..,x]] → [g ⊢ ex_bstepcong Q P A \x.Q'[..,x]] =
/ total c (cong_bstepright_impl_bstepleft _ _ _ _ _ c _)/
fn c ⇒ fn b ⇒ case c of
  | [g ⊢ par_unit] ⇒
    let [g ⊢ B] = b in [g ⊢ bsc (bs_par1 B) \x.(c_sym par_unit)]
  | [g ⊢ par_comm] ⇒
    (case b of
       | [g ⊢ bs_par1 B1] ⇒ [g ⊢ bsc (bs_par2 B1) \x.par_comm]
       | [g ⊢ bs_par2 B2] ⇒ [g ⊢ bsc (bs_par1 B2) \x.par_comm]
    )
  | [g ⊢ par_assoc] ⇒
    (case b of
       | [g ⊢ bs_par1 B] ⇒
         (case [g ⊢ B] of
            | [g ⊢ bs_par1 B'] ⇒
              [g ⊢ bsc (bs_par1 B') \x.(c_sym par_assoc)]
            | [g ⊢ bs_par2 B'] ⇒
              [g ⊢ bsc (bs_par2 (bs_par1 B')) \x.(c_sym par_assoc)]
         )
       | [g ⊢ bs_par2 B] ⇒
         [g ⊢ bsc (bs_par2 (bs_par2 B)) \x.(c_sym par_assoc)]
    )
  | [g ⊢ sc_ext_zero] ⇒ impossible b
  | [g ⊢ sc_ext_par] ⇒
    (case b of
       | [g ⊢ bs_res \x.B[..,x]] ⇒
```

```
        (case [g, x:names ⊢ B[..,x]] of
           | [g,x:names ⊢ bs_par1 B1[..,x]] ⇒
             [g ⊢ bsc (bs_par1 (bs_res \x.B1[..,x]))
             \x.(c_sym sc_ext_par)]
           | [g,x:names ⊢ bs_par2 B2[..,x]] ⇒
             let ex_bstep [g,x:names ⊢ B2[..,x]] [g ⊢ B2’] e1 e2 =
                 strengthen_bstep [g,x:names ⊢ B2[..,x]] in
             let [g,x:names ⊢ brefl] = e1 in
             let [g,x:names, z:names ⊢ prefl] = e2 in
                 [g ⊢ bsc (bs_par2 B2’) \x.(c_sym sc_ext_par)]
        )
     %| [g ⊢ bs_open \x.F[..,x]] ⇒ previously proved in section 3.3.2
    )
  | [g ⊢ sc_ext_res] ⇒
    (case b of
       | [g ⊢ bs_res \x.B[..,x]] ⇒
        (case [g,x:names ⊢ B[..,x]] of
           | [g,x:names ⊢ bs_res \y.B’[..,x,y]] ⇒
             [g ⊢ bsc (bs_res \y.(bs_res \x.B’[..,x,y])) \x.sc_ext_res]
           | [g,x:names ⊢ bs_open \y.F[..,x,y]] ⇒
             [g ⊢ bsc (bs_open \y.(fs_res \x.F[..,x,y])) \x.c_ref]
        )
       | [g ⊢ bs_open \x.F[..,x]] ⇒
        let [g,x:names ⊢ fs_res \y.F’[..,x,y]] = [g,x:names ⊢ F[..,x]]
        in [g ⊢ bsc (bs_res \y.(bs_open \x.F’[..,x,y])) \x.c_ref]
    )
  | [g ⊢ c_in \x.C1[..,x]] ⇒
    let [g ⊢ bs_in] = b in [g ⊢ bsc bs_in \x.(c_sym C1[..,x])]
  | [g ⊢ c_out C1] ⇒ impossible b
  | [g ⊢ c_par C1] ⇒
    (case b of
       | [g ⊢ bs_par1 B1] ⇒
         let [g ⊢ bsc B2 \x.C2[..,x]] =
             cong_bstepright_impl_bstepleft [g ⊢ C1] [g ⊢ B1] in
             [g ⊢ bsc (bs_par1 B2) \x.(c_par C2[..,x])]
       | [g ⊢ bs_par2 B1] ⇒
        [g ⊢ bsc (bs_par2 B1) \x.(c_par (c_sym C1[..]))]
    )
  | [g ⊢ c_res \x.C[..,x]] ⇒
    (case b of
       | [g ⊢ bs_res \x.B[..,x]] ⇒
        let [g,x:names ⊢ bsc B’[..,x] \y.C’[..,x,y]] =
            cong_bstepright_impl_bstepleft [g,x:names ⊢ C[..,x]]
            [g,x:names ⊢ B[..,x]] in
            [g ⊢ bsc (bs_res \x.B’[..,x]) \y.(c_res \x.C’[..,x,y])]
       | [g ⊢ bs_open \x.F[..,x]] ⇒
        let [g,x:names ⊢ fsc F’[..,x] C’[..,x]] =
            cong_fstepright_impl_fstepleft [g,x:names ⊢ C[..,x]]
            [g,x:names ⊢ F[..,x]] in
            [g ⊢ bsc (bs_open \x.F’[..,x]) \x.C’[..,x]]
    )
  | [g ⊢ c_ref] ⇒ let [g ⊢ B] = b in [g ⊢ bsc B \x.c_ref]
  | [g ⊢ c_sym C’] ⇒
    let [g ⊢ D] = cong_bstepleft_impl_bstepright [g ⊢ C’] b in [g ⊢ D]
  | [g ⊢ c_trans C1 C2] ⇒
    let [g ⊢ bsc B1 \x.C2’[..,x]] =
        cong_bstepright_impl_bstepleft [g ⊢ C2] b in
    let [g ⊢ bsc B2 \x.C1’[..,x]] =
        cong_bstepright_impl_bstepleft [g ⊢ C1] [g ⊢ B1] in
        [g ⊢ bsc B2 \x.(c_trans C2’[..,x] C1’[..,x])]
;
```

Bibliography

[1] Brian E. Aydemir et al. "Mechanized Metatheory for the Masses: The PoplMark Challenge". In: *Theorem Proving in Higher Order Logics*. Ed. by Joe Hurd and Tom Melham. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 50–65. ISBN: 978-3-540-31820-0.

[2] Hendrik Pieter Barendregt. *The Lambda Calculus - its Syntax and Semantics.* Vol. 103. Studies in logic and the foundations of mathematics. North-Holland, 1985. ISBN: 978-0-444-86748-3.

[3] N. G. de Bruijn. "Telescopic Mappings in Typed Lambda Calculus". In: *Inf. Comput.* 91.2 (1991), pp. 189–204. DOI: `10.1016/0890-5401(91)90066-B`.

[4] Roy L. Crole. "Alpha Equivalence Equalities". In: *Theor. Comput. Sci.* 433 (2012), pp. 1–19. DOI: `10.1016/J.TCS.2012.01.030`.

[5] N.G de Bruijn. "Lambda Calculus Notation with Nameless Dummies: A Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392. ISSN: 1385-7258. DOI: `https://doi.org/10.1016/1385-7258(72)90034-0`.

[6] Jacob Errington, Junyoung Jang, and Brigitte Pientka. "Harpoon: Mechanizing Metatheory Interactively - (System Description)". In: *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings.* Ed. by André Platzer and Geoff Sutcliffe. Vol. 12699. Lecture Notes in Computer Science. Springer, 2021, pp. 636–648. DOI: `10.1007/978-3-030-79876-5\_38`.

[7] Murdoch Gabbay and Andrew M. Pitts. "A New Approach to Abstract Syntax with Variable Binding". In: *Formal Aspects Comput.* 13.3-5 (2002), pp. 341–363. DOI: `10.1007/S001650200016`.

[8] Robert Harper, Furio Honsell, and Gordon D. Plotkin. "A Framework for Defining Logics". In: *J. ACM* 40.1 (1993), pp. 143–184. DOI: `10.1145/138027.138060`.

[9] Daniel Hirschkoff. *A Brief Survey of the Theory of the π-Calculus.* Laboratoire de l'Informatique du Parallélisme. 2+15p. 2003.

[10] Furio Honsell, Marino Miculan, and Ivan Scagnetto. "π-Calculus in (Co)Inductive Type Theory". In: *Theor. Comput. Sci.* 253.2 (2001), pp. 239–285. DOI: `10.1016/S0304-3975(00)00095-5`.

[11] William Alvin Howard. "The Formulae-as-Types Notion of Construction". In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Ed. by Haskell Curry et al. Academic Press, 1980.

[12] Frederik Krogsdal Jacobsen et al. “The Concurrent Calculi Formalisation Benchmark”. 2024.

[13] Robbert Krebbers, Alberto Momigliano, and Brigitte Pientka. *Formalization of Programming Languages*. Assia Mahboubi and Jasmin Blanchette, forthcoming.

[14] Conor McBride. “Dependently Typed Functional Programs and their Proofs”. PhD thesis. University of Edinburgh, UK, 2000. URL: `https://hdl.handle.net/1842/374`.

[15] Robin Milner, Joachim Parrow, and David Walker. “A Calculus of Mobile Processes, II”. In: *Inf. Comput.* 100.1 (1992), pp. 41–77. DOI: `10.1016/0890-5401(92)90009-5`.

[16] Joachim Parrow. “An Introduction to the π-Calculus”. In: *Handbook of Process Algebra*. Ed. by Jan A. Bergstra, Alban Ponse, and Scott A. Smolka. North-Holland / Elsevier, 2001, pp. 479–543. DOI: `10.1016/B978-044482830-9/50026-6`. URL: `https://doi.org/10.1016/b978-044482830-9/50026-6`.

[17] Doron A. Peled. “Formal Methods”. In: *Handbook of Software Engineering*. Ed. by Sungdeok Cha, Richard N. Taylor, and Kyo Chul Kang. Springer, 2019, pp. 193–222. DOI: `10.1007/978-3-030-00262-6\_5`.

[18] Frank Pfenning and Conal Elliott. “Higher-Order Abstract Syntax”. In: *Proceedings of the ACM SIGPLAN’88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*. Ed. by Richard L. Wexelblat. ACM, 1988, pp. 199–208. DOI: `10.1145/53990.54010`.

[19] Brigitte Pientka. *Beluga Reference Guide*. `https://complogic.cs.mcgill.ca/beluga/userguide2/userguide.pdf`.

[20] Brigitte Pientka. *Mechanizing Types and Programming Languages: A Companion*. `https://complogic.cs.mcgill.ca/beluga/meta.pdf`. 2015.

[21] Brigitte Pientka and Andrew Cave. “Inductive Beluga: Programming Proofs”. In: *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*. Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. Lecture Notes in Computer Science. Springer, 2015, pp. 272–281. DOI: `10.1007/978-3-319-21401-6\_18`.

[22] Brigitte Pientka and Jana Dunfield. “Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)”. In: *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*. Ed. by Jürgen Giesl and Reiner Hähnle. Vol. 6173. Lecture Notes in Computer Science. Springer, 2010, pp. 15–21. DOI: `10.1007/978-3-642-14203-1\_2`.

[23] Davide Sangiorgi and David Walker. *The π-Calculus - a Theory of Mobile Processes*. Cambridge University Press, 2001. ISBN: 978-0-521-78177-0.

[24] Alwen Tiu and Dale Miller. “Proof Search Specifications of Bisimulation and Modal Logics for the π-Calculus”. In: *ACM Trans. Comput. Log.* 11.2 (2010), 13:1–13:35. DOI: `10.1145/1656242.1656248`.

[25] Christian Urban and Michael Norrish. “A Formal Treatment of the Barendregt Variable Convention in Rule Inductions”. In: *ACM SIGPLAN International Conference on Functional Programming, Workshop on Mechanized reasoning about languages with variable binding, MERLIN 2005, Tallinn, Estonia, September 30, 2005*. Ed. by Randy Pollack. ACM, 2005, pp. 25–32. DOI: 10.1145/1088454.1088458.