UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
PROGRAMA DE MAESTRÍA Y DOCTORADO EN CIENCIAS MATEMÁTICAS Y DE LA ESPECIALIZACIÓN EN ESTADÍSTICA APLICADA

PiULL Session Types verification using COQ

TESIS
QUE PARA OPTAR POR EL GRADO DE:
MAESTRO (A) EN CIENCIAS

PRESENTA:
Ciro Iván García López

DIRECTOR
(Dr. rer. nat.) Favio Ezequiel Miranda Perea
Facultad de Ciencias
Universidad Nacional Autónoma de México

CIUDAD DE MÉXICO 2022.

Universidad Nacional
Autónoma de México

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

PIULL Session Types verification using Coq

Ciro Iván García López

Universidad Nacional Autónoma de México
Graduate School of Mathematics
Science Faculty
Ciudad de México, México
2022

PIULL Session Types verification using Coq

Ciro Iván García López

In Partial Fulfillment of the requirements for the Degree:
Master of Science Mathematics

Advisors:
(Dr. rer. nat.) Favio Ezequiel Miranda Perea

Universidad Nacional Autónoma de México
Graduate School of Mathematics
Science Faculty
Ciudad de México, México
2022

For my parents and siblings.

Acknowledgments

First, I would like to thank my supervisor, professor Favio Ezequiel Miranda Perea, whose knowledge and advice led to success of the present work. In particular, his mathematical expertise has been a source of inspiration. I have learnt a great deal from him and found a friendly hand during the hard lockdown times. I would like to acknowledge my family, who unconditionally help me through my master studies.

I am grateful to my examiners, Ph.D. Francisco Hernández Quiroz, Ph.D. Sergio Rajsbaum, Ph.D. Lourdes del Carmen Gonzáles Huesca, Ph.D. Everardo Barcenas Patiño and Ph.D. Jorge Luis Ortega Arjona for their helpful comments and useful suggestions on this work. Any mistakes that remain are my sole responsibility. I have also benefited from discussions with the professor Lourdes. She made me many helpful suggestions that improved the quality of my work.

Also, I would like to thank the Universidad Nacional Autónoma de México and México, the places where I had many experiences and grew up. Lastly, I would like to thank the Conacyt and DGAPA PAPIIT project number IN119920, for the financial support during the process.

Contents

Introduction **11**

Introducción **14**

1 Preliminaries **17**
- 1.1 Classical Linear logic . . . 17
- 1.2 Dual Context System . . . 26
- 1.3 Pi Calculus . . . 31

2 Towards a formal verification **38**
- 2.1 Session Types . . . 38
- 2.2 Subject Reduction . . . 47
- 2.3 Bounded names and its mechanization . . . 61
- 2.4 Locally Nameless Representation . . . 65

3 Design and Coq implementation **76**
- 3.1 Interactions between operations . . . 76
- 3.2 Locally Closed and Locally Closed At . . . 78
- 3.3 Multiple Open . . . 80
- 3.4 Lists or Sets . . . 81
- 3.5 Unspoken words . . . 82

4 Final remarks **85**

A The Appendix **87**
- A.1 πULL type system rules. . . . 87
- A.2 Coq Implementation . . . 92
  - A.2.1 Database and Tactics . . . 93
  - A.2.2 Propositions in Coq . . . 94
  - A.2.3 Names . . . 95
  - A.2.4 Free Names in Coq . . . 97

A.2.5 Processes in Coq 98
A.2.6 LN Representation in Coq. 99
A.2.7 Types 101

Introduction

The notion of computation is one of the most relevant ideas for science nowadays. The formalization of this concept dates back to 1936 when Church [Ber36] proposed the λ-calculus as a theory capable of representing the idea of computable function. One consequence of the proof of the equivalence between the Turing machines and the λ-calculus [Kle36] is that there are terms within the calculus which yield run-time errors, such as wrong function calls or non-coherent functions.

Here, mathematics help to understand and analyse the weaknesses or errors, within the basis and foundations of the theories. A solution provided by mathematics is the construction of collaborations between logic, and computation, to obtain better behaviours from terms. The idea is to restrict the language and build appropriate type systems.

On the other hand, computational requirements are different nowadays. These new tasks require new models of computation. So, in 1982 Milner [Mil82] proposed the calculus of the communicating systems (CCS). CCS was an answer to the need for a theory for the study of mobile systems. Later, CCS evolved into the π-Calculus [MPW92].

Using the π-calculus it is possible to study new paradigms, such as concurrent or message-passing computing. One advantage and challenge of mathematical foundations is that it is possible to abstract the critical parts of a theory. For concurrent computing: endless waiting and deadlock.

Honda [Hon93] worked on the difficulties of concurrent programming and proposed a type system as a solution. The types intend to be aware of channels' communication evolution, produced by the interaction between processes. Nevertheless, in the system proposed by Honda, the logical foundation was not clear. Hence, Pfenning, Caires [CPT12], and Wadler [Wad12a] proposed a type system for the π-calculus using linear logic. Girard [Gir87] proposed linear logic in 1987 as a resource-based logic.

For Wadler, a system of types has a good design if there is a Curry-Howard correspondence. Hence, a similar correspondence between the π-calculus processes and linear logic is highly desirable. Additionally, it should be possible to prove the Subject Reduction (types preserving) and Progress (of the reduction relation) properties in a well-designed correspondence; because they solve the endless waiting and deadlock problems.

Then, type systems motivate the study of logic systems and their properties, they also represent the complex interaction between computation and mathematics. Additionally, thanks to this interaction it is possible to construct assistants for correct software development and formal verification of

theories, as Coq[1]. This kind of software helps in the formalization and verification of mathematical knowledge. Also, they improve the understanding and prevent failures within the theory.

Verifying a theory reveals hidden details, hypotheses, and ideas. It is usual that the community accepts these details and does not talk about them. Nevertheless, they hide relevant information for the development and comprehension of the theory. Making them explicit is a challenge in formal verification and requires a deep and conscious examination of the particular case being studying.

This work presents the formal verification of the Subject Reduction Theorem for a custom version of the πULL typing system proposed by Heuvel and Perez [vdHP20]. This version fills out some ideas and definitions, in contrast with the original one. These precisions are necessary to complete the verification.

The document is divided into four chapters.

The first chapter, we introduce the ideas and origin of linear logic and give a pragmatic motivation for the system. Also, we discuss some key results and examples. Additionally, we present dual systems as a framework to work uniformly with several logics. Then, we introduce π-calculus, its motivation, and relevant ideas. Within this part, we study each definition in detail. Also, we propose a visual representation of the processes, the study of the machine representation is easier with this visual representation. The chapter ends with two key examples within the calculus: endless waiting and deadlock.

In the second chapter, we present the type system. we start by showing the correspondence between the πULL-calculus and the ULL logic system. Next, we proceed by presenting the πULL-calculus and its rules. Then, we discuss the notions of principal cuts given by Caires, Pfenning [CPT12], and Wadler [Wad12a], which support the building of the correspondence. The construction exhibits the relation between cut eliminations and computations; this helps to define the operational semantics based on cut elimination. Next, we work on the proof of necessary lemmas and the Subject Reduction Theorem.

Also, we discuss the most common machine representations their advantages and disadvantages. In particular, this part reviews the locally nameless (LN) representation and the reasons to choose it. Also, it presents its operations, functions, predicates, and it contains a proof of the equivalence between the locally closed (lc) and locally closed at (lca) predicates. This proof is relevant due to the different problems and solutions needed to complete it successfully.

[1] https://coq.inria.fr/

In the third chapter, we discuss the challenges and differences between paper and pencil proofs and mechanized proofs. The goal is to analyze and make explicit the difficulties that arose during the work and the proposed solutions. Examples guide this discussion. Finally, in the fourt chapter we do the final remarks of the present work.

The project source code can be found at: https://github.com/ cigarcial/Tmcod

Introducción

La noción de cómputo ha sido una de las ideas más relevantes en la ciencia durante las últimas décadas. La búsqueda de una formalización para este concepto encuentra sus orígenes en 1936 cuando Church [Ber36] propone el cálculo λ como una teoría capaz de representar la noción de función. Posteriormente, como consecuencia de la equivalencia entre el cálculo λ y las máquinas de Turing [Kle36], se logra establecer que dentro del mismo hay expresiones que caen en errores de ejecución, tales como usos incorrectos de funciones o resultados que carecen de sentido.

Aquí juega un rol fundamental la matemática, sentando bases y fundamentos para el estudio, comprensión y abordaje de estas falencias o errores. Un ejemplo concreto del tipo de soluciones que se ofrece, es la construcción un puente entre la lógica y la computación para mejorar el comportamiento de las expresiones de tal manera que no se obtengan errores de ejecución. Lo anterior, se traduce en combinar restricciones sintácticas y un sistema de tipos adecuado para el lenguaje.

Por otro lado, en la actualidad las necesidades computacionales han cambiado y es imprescindible pensar en nuevos marcos de trabajo matemáticos sobre los cuales sea posible formalizar y estudiar dichos requerimientos. Así por ejemplo, Milner [Mil82] en 1982 propone el cálculo de sistemas que se comunican (CCS) como respuesta a la necesidad de estudiar la computación entre sistemas móviles que se comunican; posteriormente, esta teoría se refina y nace el cálculo π [MPW92] como marco de trabajo estándar para este tipo de sistemas.

El cálculo π abrió el camino para el estudio formal de nuevos paradigmas de computación, como el paso de mensajes o la concurrencia. Una ventaja y reto, de la abstracción matemática, es que permite establecer de manera precisa las partes críticas de una teoría; en el caso de la concurrencia, los problemas de la espera mutua (Endless waiting) o interbloqueo (Deadlock).

Honda [Hon93] trabajó en las dificultades de la programación concurrente y presentó como solución un sistema de tipos, la intención es conocer cómo evoluciona un canal a medida que los procesos interactúan y se comunican entre ellos. Sin embargo, en el sistema de Honda no era explícito el fundamento lógico que subyacía, lo que motivó a que Pfenning, Caires [CPT12] y Wadler [Wad12a], propusieran un sistema de tipos para el cálculo π utilizando la lógica lineal; un sistema lógico propuesto por Girard [Gir87] en 1987 basado en la conciencia de los recursos y con diversas aplicaciones en los últimos años.

Siguiendo las ideas de Wadler, un sistema de tipos es transparente en su diseño en la medida que se obtiene una correspondencia o isomorfismo al

estilo Curry-Howard, por lo que para la programación concurrente se buscará establecer un isomorfismo análogo entre los procesos del cálculo π y las propiedades de la lógica lineal. A su vez, a la luz de una buena correspondencia debe ser posible obtener los teoremas de Reducción del Sujeto (preservación de tipos) y Progreso (de las reglas de reducción), pues al garantizar estos resultados se resuelven la espera mutua y el interbloqueo.

Se puede pensar entonces que los sistemas de tipos motivan el estudio de las lógicas y de sus propiedades, además constituyen un ejemplo de la compleja interacción que se da entre la computación y las matemáticas. Pero no solo se limitan a esto, gracias a ellos es posible construir asistentes para el desarrollo de software confiable y la verificación formal de teorías, tal como es el caso de Coq[2]. Esta clase de software permite formalizar y verificar teorías matemáticas, mejorando la comprensión y previniendo fallos dentro de las mismas.

Al verificar una teoría, es posible observar que comúnmente se esconden detalles, argumentos, hipótesis e ideas que para los autores pueden resultar sencillos o que por consenso general son aceptados. No obstante, dichos detalles omiten información relevante para el desarrollo y comprensión de la misma. Hacerlos explícitos representa uno de los retos actuales en el campo de la verificación formal, puesto que es necesario realizar un estudio consciente y profundo, para culminar en un diseño completo y exitoso.

En este trabajo se buscará verificar formalmente el teorema de Reducción del Sujeto para una versión propia del sistema de tipos πULL (United Linear Logic) propuesto por Heuvel y Perez [vdHP20]. Esta nueva versión hace precisas algunas ideas y definiciones respecto a la formulación original, refinamiento que es necesario durante el proceso de verificación.

El documento se divide en cuatro capítulos detalladas a continuación.

En el primer capítulo, se introducen las ideas de la lógica lineal, sus orígenes y se buscará dar una motivación pragmática del sistema. Así mismo, son presentados ejemplos y resultados relevantes. También se introducen y discuten los sistemas duales, los cuales fueron propuestos como un mecanismo para conciliar los distintos marcos de trabajo lógicos. El sistema lógico que se trabajará es el ULL, quien constituye la base del sistema de tipos. En la segunda parte del primer capitulo, se introduce el cálculo π su motivación e ideas relevantes, se hace un estudio de las definiciones claves, buscando introducir y exponer cada elemento con detalle. Por otro lado, se propone una representación visual para los procesos, dicha representación permite estudiar las distintas representaciones en máquina de manera más sencilla. Se finaliza esta sección presentando dos ejemplos relevantes dentro del cálculo,

[2]https://coq.inria.fr/

uno de espera mutua e interbloqueo.

En el segundo capítulo, se estudiará el sistema de tipos, la idea es motivar una correspondencia entre el cálculo πULL y la lógica ULL. Se toma como punto de partida el cálculo πULL y las reglas que lo rigen, para luego hacer explícita la construcción del sistema de tipos siguiendo las nociones de cortes principales que introducen Caires, Pfenning [CPT12] y Wadler [Wad12a]; dicha construcción es relevante en el presente trabajo dado que establece la relación entre la eliminación de cortes y los cómputos dentro del sistema, lo que se traduce en una semántica operacional basada en la eliminación de las reglas de corte. Posteriormente, se introducen algunos lemas necesarios para culminar con la prueba del Teorema de Reducción del Sujeto.

Posteriormente, se discuten e introducen las distintas representaciones en máquina para términos con variables ligadas, exponiendo ventajas y desventajas de las tres representaciones más comunes. En particular, se discute la representación local libre de nombres (LN) y los motivos por los cuales es la técnica que sigue el trabajo. Adicionalmente, se discuten las operaciones, predicados y equivalencia de los predicados lc (cerradura local) y lca (cerradura local a un nivel dado) para la representación LN. Esta última prueba es de interés ya que su verificación es un ejemplo puntual de los retos que surgen al intentar formalizar los resultados usando un asistente.

En el tercer capítulo, se amplia la discusión sobre los retos y diferencias que surgieron durante el ejercicio de verificar la prueba de Reducción del Sujeto utilizando el asistente Coq. El objetivo es, mediante ejemplos concretos, analizar y exponer las dificultades que emergieron a lo largo del trabajo. Así mismo, discutir las soluciones propuestas.

Para finalizar, en el cuarto capítulo presentamos las observaciones finales y las conclusiones del presente trabajo.

El código fuente del trabajo puede ser consultado en: https://github.com/cigarcial/Tmcod

1

Preliminaries

In this chapter, we will introduce the concepts and theories that supports the work. Our primary goal is to introduce the ideas filling up all the conceptual gaps; this effort would help us in the Coq implementation.

1.1 Classical Linear logic

Mathematics builds its theories using deductive systems. The classical logic [vP13] system supports many of the mathematics today studied. Roughly speaking, deductive systems have two elements: connectives and rules. The connectives tell us how to build propositions and the rules how to derive coherent arguments by the interaction of connectives. In particular, in classical logic, the implication is a connective, and the Modus Ponens is an inference rule.

$$\frac{A \to B \qquad A}{B} \text{ (mp)}$$

Classical logic has one non-trivial assumption: the propositions "provide infinite resources", hence we use them any time and anywhere. For example, given a natural number n, then $n/2$ can be evaluated. In particular, if we take the 8, then 4 is half of the number, but what happened to 8 after the evaluation? We answer immediately, it keeps existing as an abstract entity.

Girard [GL86] looked at this and proposed linear logic, a logic where resources matter. The idea behind this system is that we needs and uses only the necessary resources to obtain a derivation. Girard states that in classical logic two rules can be problematic: contraction and weakening. Informally, the contraction rule state that it is possible to delete duplicate resources, and the weakening rule states that it is possible to introduce new duplicate resources. Within linear logic these rules are no longer valid.

The restriction of the contraction and weakening rules reduce expressive power in linear logic. Girard introduces the WhyNot (?) and OfCourse (!) operators to recover expressiveness.

It is necessary to point out that here we discuss only syntactic elements of linear logic. The notions concerning semantics are out of the scope of the present work, but can be found at [Gir87, GL86].

The presentation used here for the sequents follows the double side style like Troelstra [Tro91], even though the notation is the proposed by Girard originally. First, let us agree that there exists an infinite number of propositional letters $p, q, r, ...$ The following grammar generates the formulas for linear logic:

$$A, B := p \mid 1 \mid \bot \mid \top \mid 0 \mid A \otimes B \mid A \& B \mid A ⅋ B \mid A \oplus B \mid A \multimap B \mid !A \mid ?A$$

An informal description for the linear logic connectives are:

- $1, \bot$: are the multiplicative neutral elements.
- $0, \top$: are the additive neutral elements.
- $A \otimes B$: there are both resources of type A and B.
- $A \& B$: there are resources of type A and B. But we can use only one of them.
- $A ⅋ B$: there is a resource of type A or type B.
- $A \multimap B$: given a resource of type A, we can produce a resource of type B.
- $A \oplus B$: there is one resource of type A or B, but it is not known which.
- $!A$: there is an unlimited supply of resources of type A.
- $?A$: there is a consumption of a resource of type A.

Greek upper letters Γ, Δ, Ω will denote multisets of formulas. In linear logic, the sequents will be written as $\Gamma \vdash \Delta$, using the symbol $\vdash$ to reinforce the idea that the sequent is linear. If $\Gamma = \{A_1, ..., A_n\}$ and $\Delta = \{B_1, ..., B_n\}$, the meaning of the sequent $\Gamma \vdash \Delta$ is that $A_1 \otimes ... \otimes A_n \vdash B_1 ⅋ ... ⅋ B_n$. $A \equiv B$ will mean that $A \vdash B$ and $B \vdash A$. Additionally, if $\Gamma = \{A_1, ..., A_n\}$, then $!\Gamma = \{!A_1, ..., !A_n\}$.

In linear logic, the upper script $^\bot$ marks the negation, and their definition is in terms of the syntaxis [Gir95]. It is common to call the negation the dual of a proposition.

Definition 1.1. *[Gir87] The negation of a proposition is defined as follows. For each propositional letter, there is always the negation of the letter denoted by $p^\perp$. For the other cases is defined as:*

$$
\begin{array}{rlrl}
1^\perp &= \perp & \perp^\perp &= 1 \\
\top^\perp &= 0 & 0^\perp &= \top \\
(p)^\perp &= p^\perp & (p^\perp)^\perp &= p \\
(A \otimes B)^\perp &= A^\perp \parr B^\perp & (A \parr B)^\perp &= A^\perp \otimes B^\perp \\
(A \& B)^\perp &= A^\perp \oplus B^\perp & (A \oplus B)^\perp &= A^\perp \& B^\perp \\
(!A)^\perp &= ?A^\perp & (?A)^\perp &= !A^\perp
\end{array}
$$

Observe that the negation is involutive, in other words $(A^\perp)^\perp = A$. The proof follows directly from the definition of negation by induction over the formula structure. Furthermore, there is no dual for the linear implication due to its definition in terms of the multiplicative disjunction, $A \multimap B = A^\perp \parr B$.

The rules of inference, in sequent style, for the linear logic are the following [Gir87].

- Identity:

$$\frac{}{A \vdash A}\ \text{(Id)}$$

- Duals:

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, A^\perp \vdash \Delta}\ \text{(LD)} \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash A^\perp, \Delta}\ \text{(RD)}$$

- Multiplicative neutrals:

$$\frac{}{\Gamma, 1 \vdash \Delta}\ (1) \qquad \frac{}{\perp\ \vdash}\ (\perp)$$

- Additive neutrals:

$$\frac{}{\Gamma, 0 \vdash \Delta}\ (0) \qquad \frac{}{\Gamma \vdash \top, \Delta}\ (\top)$$

- Implication:

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma', B \vdash \Theta}{\Gamma, \Gamma', A \multimap B \vdash \Delta, \Theta}\ (L \multimap) \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \multimap B, \Delta}\ (R \multimap)$$

- Multiplicative disjunction:

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A \parr B \vdash \Delta, \Delta'}\ (L\parr) \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \parr B, \Delta}\ (R\parr)$$

- Multiplicative conjunction:

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \otimes B \vdash \Delta}\ (L\otimes) \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma' \vdash B, \Theta}{\Gamma, \Gamma' \vdash A \otimes B, \Delta, \Theta}\ (R\otimes)$$

- Additive disjunction:

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \oplus B \vdash \Delta}\ (L\oplus)$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \oplus B, \Delta}\ (R\oplus_1) \qquad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \oplus B, \Delta}\ (R\oplus_2)$$

- Additive conjunction:

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \& B, \Delta}\ (R\&)$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \& B \vdash \Delta}\ (L\&_1) \qquad \frac{\Gamma, B \vdash \Delta}{\Gamma, A \& B \vdash \Delta}\ (L\&_2)$$

- OfCourse:

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, !A \vdash \Delta}\ (L!) \qquad \frac{!\Gamma \vdash A, ?\Delta}{!\Gamma \vdash !A, ?\Delta}\ (R!)$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, !A \vdash \Delta}\ (\text{deb!}) \qquad \frac{\Gamma, !A, !A \vdash \Delta}{\Gamma, !A \vdash \Delta}\ (\text{cnt!})$$

- WhyNot:

$$\frac{!\Gamma, A \vdash ?\Delta}{!\Gamma, ?A \vdash ?\Delta} (L?) \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash ?A, \Delta} (R?)$$

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash ?A, \Delta} (\text{deb?}) \qquad \frac{\Gamma \vdash ?A, ?A, \Delta}{\Gamma \vdash ?A, \Delta} (\text{cnt?})$$

- Structural rules:

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'} (\text{cut})$$

$$\frac{\Gamma \vdash \Delta}{\Gamma' \vdash \Delta'} (\Gamma', \Delta' \text{ are permutations of } \Gamma, \Delta)$$

By now, it is possible to remark that linear logic has two parts: multiplicative and additive. Informally, the reason for this division comes from the rules and contexts [Gir87]. The multiplicative part needs different contexts to infer. For example, the R$\otimes$ rule needs Γ to derive A, Δ; and other resources Γ' to derive B, Θ.

In contrast, the additive part works on the same context. As in the L$\oplus$ rule which needs the same context Γ and one resource (A or B), to derive Δ. The following proposition shows two admissible rules for the system. Remember that, informally, a rule is admissible if we do not need it as a primitive rule.

Proposition 1.1. *The following rules are admissible in linear logic.*

$$\frac{}{\Gamma \vdash \bot, \Delta} \qquad \frac{}{\vdash 1}$$

Proof. Using the rules of duals and neutral elements. □

The following inferences are valid in linear logic.

1. $A \oplus B \vdash B \oplus A$:

$$\frac{\dfrac{A \vdash A}{A \vdash B \oplus A} (R\oplus) \qquad \dfrac{B \vdash B}{B \vdash B \oplus A} (R\oplus)}{A \oplus B \vdash B \oplus A} (L\oplus)$$

2. $A \otimes (B \oplus C) \vdash (A \otimes B) \oplus (A \otimes C)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{A \vdash A \quad B \vdash B}{A, B \vdash A \otimes B}\,(R\otimes)}{A, B \vdash (A \otimes B) \oplus (A \otimes C)}\,(R\oplus) \qquad \dfrac{\dfrac{A \vdash A \quad C \vdash C}{A, C \vdash A \otimes C}\,(R\otimes)}{A, C \vdash (A \otimes B) \oplus (A \otimes C)}\,(R\oplus)}{A, B \oplus C \vdash (A \otimes B) \oplus (A \otimes C)}\,(L\oplus)}{A \otimes (B \oplus C) \vdash (A \otimes B) \oplus (A \otimes C)}\,(L\otimes)$$

3. $\vdash A \multimap ((A \multimap 0) \multimap 0)$:

$$\dfrac{\dfrac{\dfrac{A \vdash A \quad 0 \vdash 0}{A, A \multimap 0 \vdash 0}\,(L \multimap)}{A \vdash (A \multimap 0) \multimap 0}\,(R \multimap)}{\vdash A \multimap ((A \multimap 0) \multimap 0)}\,(R \multimap)$$

4. $A \multimap B, B \multimap C \vdash A \multimap C$:

$$\dfrac{\dfrac{\dfrac{A \vdash A \quad B \vdash B}{A \multimap B, A \vdash B}\,(L \multimap) \quad C \vdash C}{A \multimap B, B \multimap C, A \vdash C}\,(L \multimap)}{A \multimap B, B \multimap C \vdash A \multimap C}\,(R \multimap)$$

5. $!(!A \multimap B) \vdash !A \multimap !B$:

$$\dfrac{\dfrac{\dfrac{\dfrac{!A \vdash !A \quad B \vdash B}{!A \multimap B, !A \vdash B}\,(\mathrm{L}\multimap)}{!(!A \multimap B), !A \vdash B}\,(\mathrm{L}!)}{!(!A \multimap B), !A \vdash !B}\,(\mathrm{R}!)}{!(!A \multimap B) \vdash !A \multimap !B}\,(\mathrm{R}\multimap)$$

6. $!(A \& B) \equiv !A \otimes !B$. This derivation has two parts:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A \vdash A}{A \& B \vdash A}\,(L\&)}{!(A \& B) \vdash A}\,(L!)}{!(A \& B) \vdash !A}\,(R!) \qquad \dfrac{\dfrac{\dfrac{B \vdash B}{A \& B \vdash B}\,(L\&)}{!(A \& B) \vdash B}\,(L!)}{!(A \& B) \vdash !B}\,(R!)}{!(A \& B), !(A \& B) \vdash !A \otimes !B}\,(R\otimes)}{!(A \& B) \vdash !A \otimes !B}\,(\mathrm{cnt}!)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A \vdash A}{!A \vdash A}\,(L!)}{!A, !B \vdash A}\,(\text{deb!}) \quad \dfrac{\dfrac{B \vdash B}{!B \vdash B}\,(L!)}{!A, !B \vdash B}\,(\text{deb!})}{!A, !B \vdash A \& B}\,(R\&)}{!A, !B \vdash !(A \& B)}\,(R!)}{!A \otimes !B \vdash !(A \& B)}\,(L\otimes)$$

The Cut Elimination Theorem holds in linear logic. It implies the subformula property and decidability of the logic.

Theorem 1.1. *For a given proof in linear logic, there is cut-free proof. In other words, there is a derivation that does not use the cut rule.*

Proof. See [GL86, Tro91]. □

In particular, Girard states that a consequence of Theorem 1.1 is the impossibility to derive the sequent $A, B \vdash A$ in linear logic [GL86]: suppose that there is a derivation for the sequent, we can take one that has no cut rules and are of minimum height. Then, the last rule applied must be a permutation or identity rule, because cannot be a rule that introduces a connector. But, both of them produce a contradiction; the sequent is not initial and was of minimum height. The result agrees with the read of the sequents and connectives, resource B cannot just disappear.

The following is an important lemma for linear logic. It is known as the inversion lemma.

Proposition 1.2. *The following holds in linear logic:*

- *If* $\Gamma, A \otimes B \vdash \Delta$ *then* $\Gamma, A, B \vdash \Delta$.
- *If* $\Gamma, A \oplus B \vdash \Delta$ *then* $\Gamma, A \vdash \Delta$ *and* $\Gamma, B \vdash \Delta$.
- *If* $\Gamma \vdash A \& B, \Delta$ *then* $\Gamma \vdash A, \Delta$ *and* $\Gamma \vdash B, \Delta$.
- *If* $\Gamma \vdash A \multimap B, \Delta$ *then* $\Gamma, A \vdash B, \Delta$.
- *If* $\Gamma \vdash A \parr B, \Delta$ *then* $\Gamma \vdash A, B, \Delta$.
- *If* $\Gamma, A^{\perp} \vdash \Delta$ *then* $\Gamma \vdash A, \Delta$.
- *If* $\Gamma \vdash A^{\perp}, \Delta$ *then* $\Gamma, A \vdash \Delta$.

Proof. For all cases, apply the cut rule. For example, for the linear implication:

$$\dfrac{\dfrac{A \vdash A \qquad B \vdash B}{A, A \multimap B \vdash B}\,(L \multimap) \qquad \Gamma \vdash A \multimap B, \Delta}{\Gamma, A \vdash B, \Delta}\,(\text{cut})$$

□

We have intuitionistic logic as a linear logic subsystem. Before discussing the immersion, it is necessary to fix a sequent system for intuitionistic logic. Here we use the system exposed by Von Plato in [vP13]. First, let's fix a translation between the two systems, known as Girard translation [Pai90]:

$$\begin{aligned} p^\circ &= p \\ \bot^\circ &= 0 \\ (A \wedge B)^\circ &= A^\circ \& B^\circ \\ (A \vee B)^\circ &= !(A^\circ) \oplus !(B^\circ) \\ (A \to B)^\circ &= !(A^\circ) \multimap (B^\circ) \\ (\neg A)^\circ &= !(A^\circ) \multimap 0 \end{aligned}$$

Remember that if $\Gamma = \{A_1, ..., A_n\}$ then $!\Gamma = \{!A_1, ..., !A_n\}$.

Theorem 1.2. *If* $\Gamma \vdash A$*, then* $!\Gamma^\circ \vdash A^\circ$*.*

Proof. By induction on the height of the derivation, and is similar to the proof done by De Paiva in [Pai90]. The base cases are when the derivation $\Gamma \vdash A$ has height one, hence the sequence is initial:

- If $\Gamma \vdash A$, then it is true that $A \in \Gamma$, therefore $!A^\circ \in !\Gamma^\circ$ and:

$$\dfrac{\dfrac{!\Gamma^\circ, A^\circ \vdash A^\circ}{!\Gamma^\circ, !A^\circ \vdash A^\circ}\,(L!)}{!\Gamma^\circ \vdash A^\circ}\,(\text{cnt!})$$

- If $\bot \in \Gamma$, then $0 \in !\Gamma^\circ$ and using the rule for zero $!\Gamma^\circ \vdash A^\circ$.

Now suppose that for all the proofs of height n or less the result is valid, and that there is a derivation of height $n+1$ whose last rule is:

- $R\wedge$. The desire result is $!\Gamma^\circ \vdash A^\circ \& B^\circ$, and the intuitionistic proof has the structure:

$$\dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,(R\wedge)$$

By induction hypothesis $!\Gamma^\circ \vdash A^\circ$, and $!\Gamma^\circ \vdash B$, hence:

$$\frac{!\Gamma^\circ \vdash A^\circ \qquad !\Gamma^\circ \vdash B^\circ}{!\Gamma^\circ \vdash A^\circ \& B^\circ}\ (R\&)$$

- $L\wedge$. The goal is $!\Gamma^\circ, !(A^\circ \& B^\circ) \vdash C$, and the intuitionistic proof has the structure:

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C}\ (L\wedge)$$

By induction hypothesis $!\Gamma^\circ, !A^\circ, !B^\circ \vdash C^\circ$, therefore:

$$\frac{\dfrac{!\Gamma^\circ, !A^\circ, !B^\circ \vdash C^\circ}{!\Gamma^\circ, !A^\circ \otimes !B^\circ \vdash C^\circ}\ (L\otimes) \qquad !(A^\circ \& B^\circ) \vdash !A^\circ \otimes !B^\circ}{!\Gamma^\circ, !(A^\circ \& B^\circ) \vdash C^\circ}\ (\text{cut})$$

The right upper sequent was derived earlier.

- $R\vee$. The conclusion should be $!\Gamma^\circ \vdash !A^\circ \oplus !B^\circ$, and the intuitionistic proof structure is:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ (R\vee)$$

By induction hypothesis $!\Gamma^\circ \vdash A^\circ$, and as conclusion:

$$\frac{\dfrac{!\Gamma^\circ \vdash A^\circ}{!\Gamma^\circ \vdash !A^\circ}\ (R!)}{!\Gamma^\circ \vdash !A^\circ \oplus !B^\circ}\ (R\oplus)$$

- $L\vee$. Now the desired conclusion is $!\Gamma^\circ, !(!A^\circ \oplus !B^\circ) \vdash C^\circ$, and the intuitionistic proof structure is:

$$\frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C}\ (L\vee)$$

By induction hypothesis $!\Gamma^\circ, !A^\circ \vdash C^\circ$, $!\Gamma^\circ, !B^\circ \vdash C^\circ$, therefore:

$$\frac{!\Gamma^\circ, !A^\circ \vdash C^\circ \qquad !\Gamma^\circ, !B^\circ \vdash C^\circ}{!\Gamma^\circ, !A^\circ \oplus !B^\circ \vdash C^\circ}\ (L\otimes)$$

- $R \rightarrow$. The goal is to prove that $!\Gamma \vdash !A^\circ \multimap B^\circ$ and the intuitionistic proof structure is:

$$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}\,(R \rightarrow)$$

By induction hypothesis $!\Gamma^\circ, !A^\circ \vdash B^\circ$, hence:

$$\dfrac{!\Gamma^\circ !A^\circ \vdash B^\circ}{!\Gamma^\circ \vdash !A^\circ \multimap B}\,(R \multimap)$$

- $L \rightarrow$. It is required that $!\Gamma^\circ, !(!A^\circ \multimap B^\circ) \vdash C^\circ$, and the intuitionistic proof structure is:

$$\dfrac{\Gamma, A \rightarrow B \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \rightarrow B \vdash C}\,(L \rightarrow)$$

By induction hypothesis $!\Gamma^\circ, !(!A^\circ \multimap B^\circ) \vdash A^\circ$, $!\Gamma^\circ, !B^\circ \vdash C^\circ$ and:

$$\dfrac{\dfrac{\dfrac{\dfrac{!\Gamma^\circ, !(!A^\circ \multimap B^\circ) \vdash A^\circ}{!\Gamma^\circ, !(!A^\circ \multimap B^\circ) \vdash !A^\circ}\,(R!) \qquad !\Gamma^\circ, !B^\circ \vdash C^\circ}{!\Gamma^\circ, !\Gamma^\circ, !(!A^\circ \multimap B^\circ), !A^\circ \multimap !B^\circ \vdash C^\circ}\,(L \multimap) \qquad \dfrac{}{!(!A^\circ \multimap B^\circ) \vdash !A^\circ \multimap !B^\circ}}{!\Gamma^\circ, !\Gamma^\circ, !(!A^\circ \multimap B^\circ), !(!A^\circ \multimap B^\circ) \vdash C^\circ}\,(\text{cut})}{!\Gamma^\circ, !(!A^\circ \multimap B^\circ) \vdash C^\circ}\,(\text{cnt!})$$

The sequent without a tag was derived earlier.

In all the cases, the required conclusion follows. Hence, by the principle of induction, the result is valid. □

1.2 Dual Context System

The development of linear logic became a cornerstone to formalize computational ideas using a logical framework [GL86]. Many of the concepts developed within linear logic have applications in programming languages and computational systems design [GM94, GL86].

Nevertheless, it is common to combine different frameworks and works. Hence, the community needs a theory that unifies the logical systems. Girard

[Gir93] proposed the Unity of Logic as a sequent theory in which it is possible to work with two logical systems.

In Girard formulation, the sequents are of the form $\Gamma; \Gamma' \vdash \Delta'; \Delta$. The contexts Γ, Δ work linearly and Γ', Δ' work classically. The meaning of the sequent is that there is a linear derivation of $\Gamma, !\Gamma' \vdash ?\Delta', \Delta$. Within this theory, the semicolon (;) separates the linear and classical parts.

With different behaviours for the propositions. It is common to propose different versions of the rules. For example, depending on the position of A there are different cut rules. Girard [Gir93] the following pure linear and linear-classic rules:

$$\frac{\Gamma; \Gamma' \vdash \Delta'; \Delta, A \qquad A, \Lambda; \Gamma' \vdash \Delta'; \Pi}{\Gamma, \Lambda; \Gamma' \vdash \Delta'; \Delta, \Pi} \text{(Cut1)}$$

$$\frac{\Gamma; \Gamma' \vdash \Delta', A; \Delta \qquad A; \Gamma' \vdash \Delta';}{\Gamma; \Gamma' \vdash \Delta'; \Delta} \text{(Cut2)}$$

A relevant in the calculus given by Girard is the proposition ability to move from one side of the turnstile to the other. Girard's rules rely in the notion of polarities of a given formula [Gir93]. There are three polarities: positive, negative, and neutral. For example, consider a formula P with positive polarity, informally, this means that $P \equiv !P$ in linear logic. Girard introduces the following rule using positive polarities:

$$\frac{\Gamma; P, \Gamma' \vdash \Delta'; \Delta}{\Gamma, P; \Gamma' \vdash \Delta'; \Delta} \text{(Polar)}$$

Although, the notion of polarities is quite complex and requires concepts from linear logic semantics to understand them. As was discussed, this new calculus is complex and requires many rules. Girard comments about his system:

> [Gir93, p. 204] The system presented here is rather big, for the reason that we used a two-sided version to accommodate intuitionistic features more directly, and because there are classical, intuitioistic and linear connectives; last, but not least rules can split into several cases depending on polarities; the rules for disjunction, for instance, fill a whole page! But this complication is rather superficial...

The reunification of logic proposed by Girard, lead to a new research area [Bar96]. Many of the proposed dual context systems based on Girard's ideas use a subsystem of the logic, the idea is to produce systems easier to

work with. For instance, Barber [Bar96] worked with sequents of the form $\Gamma; \Delta \vdash A$, in which Γ works intuitionistic and Δ linearly. Barber also restricts the system to the linear connectives $\multimap, \otimes, !$, and 11 rules. The interaction between the intuitionistic and linear part happens in the OfCourse connector rules.

$$\frac{\Gamma; \vdash A}{\Gamma; \vdash !A}\ (!\mathrm{I}) \qquad\qquad \frac{\Gamma; \Delta_1 \vdash !A \qquad \Gamma, A; \Delta_2 \vdash B}{\Gamma; \Delta_1, \Delta_2 \vdash B}\ (!\mathrm{E})$$

These rules are easier to understand using the ideas of linear logic. For example in the first rule, if the linear context is empty, then it is possible to obtain an unlimited number of resources. One advantage, of the system given by Barber, is that it does not require complex ideas and many details of the system can be extracted of its syntax.

The systems proposed by Girard [Gir93] and Barber [Bar96] are important to understand dual context systems. The first motivates its discussion and is a guide to propose a new one; the second is closer to the system used here.

Now, we introduce a dual system that supports the type system proposed by Heuvel and Perez[vdHP20]. Heuvel and Perez called their system United Linear Logic (ULL), which is a dual context system for the linear logic. Now, we present the ULL grammar.

Definition 1.2. *The next grammar generates propositions for the ULL system:*

$$A, B := 1 \mid \bot \mid A \otimes B \mid A ⅋ B \mid !A \mid ?A$$

In ULL the sequents are of the form $\Gamma; \Delta \vdash \Lambda$ in which the formulas in Δ, Λ can be used linearly (exactly once) and the propositions in Γ an indefinite number of times (included zero). The dual is defined in the same way as Definition 1.1. The rules for the system are the following.

- Identity:

$$\frac{}{\Gamma; A, A^{\perp} \vdash \cdot}\ (\mathrm{IdL}) \qquad\qquad \frac{}{\Gamma; A \vdash A}\ (\mathrm{IdR})$$

- Neutral elements:

$$\frac{\Gamma; \Delta \vdash \Lambda}{\Gamma; \Delta, 1 \vdash \Lambda}\ (1\mathrm{L}) \qquad\qquad \frac{}{\Gamma; \cdot \vdash 1}\ (1\mathrm{R})$$

$$\frac{}{\Gamma; \bot \vdash \cdot}\ (\bot L) \qquad \frac{\Gamma; \Delta \vdash \Lambda}{\Gamma; \Delta \vdash \Lambda, \bot}\ (\bot R)$$

- Multiplicative conjunction:

$$\frac{\Gamma; \Delta, A, B \vdash \Lambda}{\Gamma; \Delta, A \otimes B \vdash \Lambda}\ (\otimes L)$$

$$\frac{\Gamma; \Delta \vdash \Lambda, A \qquad \Gamma; \Delta' \vdash \Lambda', B}{\Gamma; \Delta, \Delta' \vdash \Lambda, \Lambda', A \otimes B}\ (\otimes R)$$

- Multiplicative disjunction:

$$\frac{\Gamma; \Delta, A \vdash \Lambda \qquad \Gamma; \Delta', B \vdash \Lambda'}{\Gamma; \Delta, \Delta', A \parr B \vdash \Lambda, \Lambda'}\ (\parr L)$$

$$\frac{\Gamma; \Delta \vdash \Lambda, A, B}{\Gamma; \Delta \vdash \Lambda, A \parr B}\ (\parr R)$$

- OfCourse:

$$\frac{\Gamma, A; \Delta \vdash \Lambda}{\Gamma; \Delta, !A \vdash \Lambda}\ (!L) \qquad \frac{\Gamma; \cdot \vdash A}{\Gamma; \cdot \vdash !A}\ (!R)$$

- WhyNot:

$$\frac{\Gamma; A \vdash \cdot}{\Gamma; ?A \vdash \cdot}\ (?L) \qquad \frac{\Gamma, A; \Delta \vdash \Lambda}{\Gamma; \Delta \vdash \Lambda, ?A^{\bot}}\ (?R)$$

- Copy rules:

$$\frac{\Gamma, A; \Delta, A \vdash \Lambda}{\Gamma, A; \Delta \vdash \Lambda}\ (\text{CopyL}) \qquad \frac{\Gamma, A; \Delta \vdash \Lambda, A^{\bot}}{\Gamma, A; \Delta \vdash \Lambda}\ (\text{CopyR})$$

- Structural rules:

$$\frac{\Gamma; \Delta, A \vdash \Lambda \qquad \Gamma; \Delta', A^{\bot} \vdash \Lambda'}{\Gamma; \Delta, \Delta' \vdash \Lambda, \Lambda'}\ (\text{CutL})$$

$$\frac{\Gamma; \Delta \vdash \Lambda, A \qquad \Gamma; \Delta', A \vdash \Lambda'}{\Gamma; \Delta, \Delta' \vdash \Lambda, \Lambda'}\ (\text{CutR})$$

$$\frac{\Gamma; \cdot \vdash A \qquad \Gamma, A; \Delta \vdash \Lambda}{\Gamma; \Delta \vdash \Lambda} \text{(Cut!)}$$

$$\frac{\Gamma; A^{\perp} \vdash \cdot \qquad \Gamma, A; \Delta \vdash \Lambda}{\Gamma; \Delta \vdash \Lambda} \text{(Cut?)}$$

Here the linear implication is not considered as a primitive connector. Instead, it is defined using the multiplicative disjunction. To understand better the dual context system, some derivations are presented.

Proposition 1.3. *The following rules are admissible in the dual system.*

$$\frac{\Gamma; \Delta \vdash \Delta, A}{\Gamma; \Delta, A^{\perp} \vdash \Delta} \qquad \frac{\Gamma; \Delta, A \vdash \Delta}{\Gamma; \Delta \vdash \Delta, A^{\perp}}$$

$$\frac{}{A; \cdot \vdash A} \qquad \frac{}{A; A^{\perp} \vdash \cdot}$$

Proof. For the first, the proof is a combination of a cut rule and one identity rule:

$$\frac{\Gamma; \Delta \vdash \Delta, A \qquad \dfrac{}{\Gamma; A, A^{\perp} \vdash \cdot} \text{(IdL)}}{\Gamma; \Delta, A^{\perp} \vdash \Delta} \text{(CutR)}$$

For the last, the proof is an application of the copy rule:

$$\frac{\dfrac{}{A; A, A^{\perp} \vdash \cdot} \text{(IdL)}}{A; A^{\perp} \vdash \cdot} \text{(CopyL)}$$

The other two derivations are similar. □

Some linear logic results extend naturally.

Proposition 1.4. *The following holds in the dual system:*

- *If* $\Gamma; \Delta, A \otimes B \vdash \Lambda$ *then* $\Gamma; \Delta, A, B \vdash \Lambda$.
- *If* $\Gamma; \Delta \vdash A \parr B, \Lambda$ *then* $\Gamma; \Delta \vdash A, B, \Lambda$.

Proof. For the multiplicative conjunction, we can obtain it as follows:

$$\frac{\dfrac{\Gamma; A \vdash A \qquad \Gamma; B \vdash B}{\Gamma; A, B \vdash A \otimes B} (\otimes\text{R}) \qquad \Gamma; \Delta, A \otimes B \vdash \Lambda}{\Gamma; \Delta, A, B \vdash \Lambda} \text{(CutR)}$$

The $\parr$ case is similar to the previous one. □

As expected, cut rules can be eliminated. Unfortunately, the proof is complex and requires simultaneous induction on each cut rule. Pfenning proves this result for a similar system [Pfe95].

Theorem 1.3. *In ULL can be eliminated cut rules.*

Proof. See [Pfe95]. □

1.3 Pi Calculus

In the early years of computer science, there was a necessity of formalizing the vague ideas of "algorithm". As an answer to this question three independent models arose, which explain the mathematical behavior and computational content of this idea: Turing and the Turing machines, Gödel and the recursive functions, and Church with the lambda calculus. Many mathematical functions and algorithms have a representation using these three models.

In later years, computer machines evolved rapidly, and their complexity grew exponentially. In particular, with the initiation of communication between computers and the creation of networks, a new way to build and interact with the machines appeared, but the new world was no longer representable using classical models. Robin Milner comments the following.

> [Mil99, p. 3] But nowadays most computations involves interaction - and therefore involve systems with components which are concurrently active. Computer science must therefore rise to the challenge of defining an underlying model, with a small number of basic concepts, in terms of which interactions behavior can be rigorously described.

The need for a theory that helped with this work was evident. Robert Milner devised a new formalization for this, the π-calculus, or the calculus of mobility. This section intends to introduce a fragment of the π-calculus. The key ideas for this presentation follow the textbooks Milner [Mil99] and Sangiorgi [SW03].

Assume that there is an infinite set of names $x, y, z, ...$ Names represent communication channels. The action of the calculus, denoted by π, are constructed using names as follows.

$$\pi = x(y) \mid x\langle y \rangle$$

With the actions, it is possible to construct the processes as follows.

$$P, Q = \boldsymbol{\theta} \mid \pi.P \mid P|Q \mid !P \mid \nu x.P$$

Concerning the processes, it is worthy to point out some observations. The non deterministic operator $(+)$ is not part of this presentation. The second is that we use the symbol $\boldsymbol{\theta}$ instead of 0 to denote the process zero, avoiding confusion with the natural number zero. The third is that $x(y)$ works similar to the λ in Lambda calculus, it binds the occurences of y within a process, but the difference is that additionally represents an action in π-calculus. The actions and processes interpretations are:

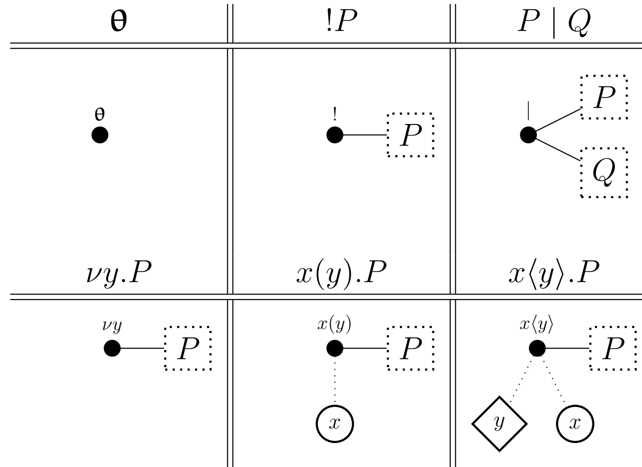
- $x(y)$, receive y along the name x.
- $x\langle y\rangle$, send y along the name x.
- $P \mid Q$, parallel run of processes P and Q. They proceed independently but can interact via shared names.
- $!P$, replication of the process P.
- $\nu z.P$, restriction of the name z to the process P.

The following are some examples of processes in π-calculus and their informal meaning.

- $x(y).P \mid x\langle z\rangle.Q$. The process P needs a resource y over the channel x. Independently Q offers a resource z over the same channel x.
- $!x(y).P$. A replicate process P that is always accepting a petition over x.

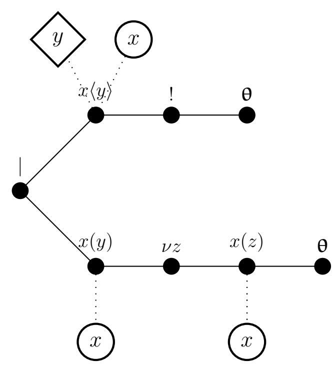
We could say that the grammar of the π-calculus is limited, but it is worthwhile to mention that this calculus intends to study the communication between the processes and not any process itself. Then the internal representation and operations that a process executes are abstracted in this calculus.

Now, it is convenient to have a visual representation for the terms of the π-calculus, this helps to understand the machine representation. We propose this visual representation, it finds inspiration in the De Bruijn [de 72] for λ-calculus. Given a process, it will be drawn as a rooted tree constructed reading the term from left to right using the following inductive definition.

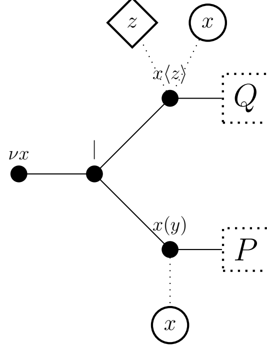


In our representation there are two types of nodes: connector nodes, we represent it with a bullet labelled with the connector, and them represent the grammar terms; and name nodes. For the actions νy and $x(y)$, y is not represented; because it denotes a binding and not a name. Furthermore, we represent channel x in actions $x(y).P$ and $x\langle y\rangle.P$ with a circle; and the channel y in output processes as a diamond. Consider the following examples and their representations:

- $(x\langle y\rangle.!\boldsymbol{\theta}) \mid (x(y).\nu z.x(z).\boldsymbol{\theta})$:



- $\nu x.(x(y).P \mid x\langle z\rangle.Q)$:



In $\nu z.P$, $x(y).Q$, z, y are bind in P and Q respectively. If a name is not bound it is said to be free. The following recursive definition, $fn(P)$, computes the set of free names for a process P [SW03].

$$
\begin{aligned}
fn(\mathbf{0}) &= \{\} \\
fn(P \mid Q) &= fn(P) \cup fn(Q) \\
fn(!P) &= fn(P) \\
fn(x\langle y\rangle.P) &= \{x, y\} \cup fn(P) \\
fn(x(y).P) &= (\{x\} \cup fn(P)) \setminus \{y\} \\
fn(\nu x.P) &= fn(P) \setminus \{x\}
\end{aligned}
$$

It is convenient to say that x is fresh in a process P if $x \notin fn(P)$. Now, the names in the π calculus intend to receive and send other names. For this, we need a precise notion of name substitution. Any substitution must be made with some restrictions in mind, just as in the λ-calculus, it is mandatory that the capture of names by binders does not occur.

Definition 1.3. *[SW03] A substitution is a function from names to names that is the identity except on a finite set. Then if $\{x_1, ..., x_n\}$ is the set with images $\{y_1, ..., y_n\}$ (respectively), the function can be denoted as $\{y_1, ..., y_n/x_1, ..., x_n\}$.*

If the names $y_1, ..., y_n$ do not occur in the process P, $P\{y_1, ..., y_n/x_1, ..., x_n\}$ means that each free occurrence of x_i in P is replaced with y_i.

We make the substitution with precaution when it changes a bound channel. The idea is to choose an α-equivalent (Informally, we can change bound names without changing a term's meaning. We will define this relation below.) process to exchange the bound channel, and then make the substitution. We can make this because any bound name work within a local scope, i.e., only inside the process. Hence, if the name is changed, the behaviour of the process must not change.

Definition 1.4. *A change of a bound name in a process P occurs when a subterm $x(z).Q$ in P is substituted by the subterm $x(w).Q\{w/z\}$ or a subterm $\nu z.Q$ for $\nu w.(Q\{w/z\})$.*

Two processes: P, Q are α-equivalent ($P =_\alpha Q$) if Q can be obtained from P by a finite number of bound names changes.

The next pairs of the process are α-equivalent.

- $\nu z.P$ and $\nu x.(P\{x/z\})$.
- $\nu x.\nu y.(x\langle y\rangle.\mathbf{0} \mid x(z).\mathbf{0})$ and $\nu z.\nu y.(z\langle y\rangle.\mathbf{0} \mid z(w).\mathbf{0})$.

The definition of a congruence relation over processes requires a concept that allows to focus on a subterm in which a replacement can be done.

Definition 1.5. *[SW03] A context is obtained when the hole, denoted by $[\cdot]$, replaces one occurrence of $\mathbf{0}$ in the process.*

Here should be pointed out that a context has one hole, no more. Some examples of contexts are:

- $[\cdot]$, the trivial context.
- $\nu x.\mathbf{0} \mid [\cdot]$.
- $z(y).(y\langle w\rangle.\mathbf{0} \mid !\nu y.[\cdot])$.

As in the λ-calculus, the task of filling the hole in a context $C[\cdot]$ with a process P, is denoted with $C[P]$.

For example, filling the third example with $\nu x.x(y).\mathbf{0}$ yields $z(y).(y\langle w\rangle.\mathbf{0} \mid !\nu y.\nu x.x(y).\mathbf{0})$. As we said, the contexts motivate congruences over processes.

Definition 1.6. *An equivalence relation ($\cong$) on processes is a congruence if $P \cong Q$ implies $C[P] \cong C[Q]$ for every context C.*

Now is time to talk about how to compute using π-calculus. Computation is done using two mechanisms: congruence on terms and terms reduction. Congruence associates processes that behave in the same way but are syntactically different. Reduction is about the interaction between processes.

To illustrate, consider the process $P \mid \mathbf{0}$, its informal description is two processes that are acting independently but with the particularity that the second does nothing. Then the behavior of this process is the same as P, and it should not be a difference if P is preferred rather than $P \mid \mathbf{0}$. The following definition captures the essence of these ideas.

Definition 1.7. *[SW03] Given two processes P, Q it will be said that are structural congruent if we can transform one into the other using the following equations (in either direction):*

$$\begin{array}{rr} P \equiv P \mid \mathbf{0} & P \mid (Q \mid R) \equiv (P \mid Q) \mid R \\ P \mid Q \equiv Q \mid P & P \mid !P \equiv !P \\ \nu x.\mathbf{0} \equiv \mathbf{0} & P \equiv Q, \textit{ if } P =_\alpha Q \\ \nu x.\nu y.P \equiv \nu y.\nu x.P & P \mid \nu x.Q \equiv \nu x.(P \mid Q), \textit{ if } x \notin fn(P) \end{array}$$

The structural congruence ($\equiv$) relation is the smallest congruence on processes that satisfies these transformations.

The following proposition is an easy observation from the definition.

Proposition 1.5. *Given a process P and a name x such that $x \notin fn(P)$, then $\nu x.P \equiv P$.*

Proof. As $x \notin fn(P)$ then, $P \equiv P \mid \mathbf{0} \equiv P \mid \nu x.\mathbf{0} \equiv \nu x.(P \mid \mathbf{0}) \equiv \nu x.P$. Observe that the notion of congruence is present in the second and fourth equations. □

As we observed, processes term reduction captures the interaction between processes, in a similar way to β-reductions in λ calculus. With this notion, processes are capable of exchange information and communicating between themselves.

Definition 1.8. *Reduction of terms ($\longrightarrow$) is defined by the following rules:*

$$\frac{}{x\langle y\rangle.P \mid x(z).Q \longrightarrow P \mid Q\{y/z\}} \textit{(react)}$$

$$\frac{P_1 \equiv P_2 \qquad Q_1 \equiv Q_2 \qquad P_1 \longrightarrow Q_1}{P_2 \longrightarrow Q_2} \textit{(struct)}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \textit{(par)} \qquad \frac{P \longrightarrow Q}{\nu x.P \longrightarrow \nu x.Q} \textit{(res)}$$

The next result follows easily from the definitions.

Proposition 1.6. *The following reduction rule is admissible for the reduction of terms:*

$$\frac{}{x\langle y\rangle.P \mid !x(z).Q \longrightarrow P \mid Q\{y/z\} \mid !x(z).Q} \textit{(rep)}$$

Proof. It follows from the rule of structural congruence and the struct reduction rule. □

Finally, we present two key examples from π-calculus, which are instances of runtime errors. They illustrate that within the π-calculus some terms have bad behavior; i.e., cannot be reduced or their reduction stucks. As Dal Lago et. al. [DLdVMY19] explain, many type systems finds their motivation in this kind of errors.

Example 1.1. This error arises when process A expects an input/output over a channel, although no process can execute the request and answer. A π-calculus term of this sort is:

$$x(y).P \mid w\langle z\rangle.Q$$

Here P will wait for input over x, but Q cannot fulfill this requirement as it uses the channel w. As a consequence, P will be waiting forever.

Example 1.2. The second kind error that is frequent when dealing with message passing systems is the deadlock. This behavior arises when process A expects some response from B to continue its execution, whereas B expects a response from A. As a consequence neither A nor B continue executing, due to a cycle dependency. A term of this kind in the π calculus is:

$$x(y).w\langle z\rangle.P \mid w(y).x\langle z\rangle.Q$$

Here P ask Q for a resource on x, Q knows the answer for the request but before giving it, needs another resource over w that P has. Inevitably, neither P nor Q can advance; they are stuck.

In this chapter, we introduced the ideas of linear logic. We discussed some foundational aspects of the logic, their origin and motivation. As it was stated, linear logic plays an important role in computational systems design; hence, it is relevant to know and study the logic from a mathematical point of view and its relation with other logical systems. Also, this chapter has the fundamental ideas behind the dual context systems. The ULL system is necessary for the study of the typing system, it is the logic system behind the types.

As part of the presentation, we stated the inversion lemmas and the theorems of cut elimination. These are significant results, due to their complexity and richness from the mathematical point of view. Lastly, we presented the π-calculus ideas, motivation and origin. All these notions will help us to define our version of the typed calculus. Also, we will review these results from the typing point of view where their have a special meaning.

2

Towards a formal verification

In this chapter we present our version of the type system. We will use the notion of principal cuts to verify that the system's rules are correct. Then, we will discuss the chosen terms representation.

2.1 Session Types

Previously, we presented π-calculus. This theory describes the communication between processes. Besides, we discussed some examples within the system. In particular, Examples 1.1, 1.2 are of interest, both refer to the two most common errors when talking about communication in computation.

Searching error-free systems, Honda develops the first type system for the concurrent programming. The idea is to seek a system in which, as a consequence, there is a deadlock-free behavior [Hon93].

The best way to introduce session types is by discussing first, the parallel that Wadler makes between them and the λ-calculus [Wad12a, Wad12b]. It is known that within untyped λ-calculus one can codify a term whose reduction never stops, for example, $(\lambda x.(xx))(\lambda x.(xx))$; these kind of terms are called non-halting. The non-halting functions are a problem for λ-calculus since they refer to an infinite computational process.

To overcome this problem, type systems were proposed for the λ-calculus. These systems filter out those problematic terms, and focus on terms that have good behavior [SU06].

Later, Curry, and Howard [SU06] encountered a result that relates the logic and the computation. They found that there is a correspondence between intuitionistic logic and typed λ-calculus:

propositions	as	types
proofs	as	programs
normalisation of proofs	as	evaluation of programs

This result is known as the Curry-Howard isomorphism and provides a variety of consequences. As it is common in mathematics, the ideas of Curry

and Howard inspire some other authors, and today it is common to study the typed systems searching for similar correspondences.

Nowadays, some works follow these ideas for the theory of session types [Wad12a, CPT12, vdHP20, Wad12b]. In particular, Wadler states that the correspondence in session types theories is the following.

propositions	as	session types
proofs	as	processes
cut elimination	as	communication

Yet, there is a radical difference between the Curry-Howard correspondence and the theory for session types. In Curry-Howard correspondence, types are for functions and data; in session types, types tell us how the communication evolves within the communication channels.

Session types theories aim for communication systems that are free of runtime and deadlock errors. Henceforth, the goal is to verify that the typing system satisfies the Subject Reduction [Ton15]. Heuvel and Perez discussed a type system with these properties in [vdHP20], and we will use their system as a guide.

Here, we are going to present our proper type system. We intend to construct the best possible design that makes the verification easy. The idea is to reconstruct the type system from scratch, explaining the meaning of each part and its relation with the other components. The presentation given here finds its inspiration in the work of Caires et. al., [CPT12].

One keystone goal of the present work is to complete the verification of the Subject Reduction Theorem on Coq. In this direction, we want to present all the system details, which will make the verification straightforward.

We will use the following π-calculus grammar variation.

Definition 2.1. *The following grammar generates the processes for the type system:*

$$P, Q = \boldsymbol{\theta} \mid [x \leftrightarrow y] \mid P|Q \mid \nu x.P \mid x\langle y\rangle.P \mid x(y).P \mid x().P \mid !x(y).P \mid x\langle\rangle.\boldsymbol{\theta}$$

Here x, y represent channel names (strings).

We will abuse of notation and call πULL the above calculus and the type system presented below.

The processes constructed using π-calculus grammar keep their original aim. For example, $!x(y).P$ means the replicated input for P. For the new ones, their meaning is:

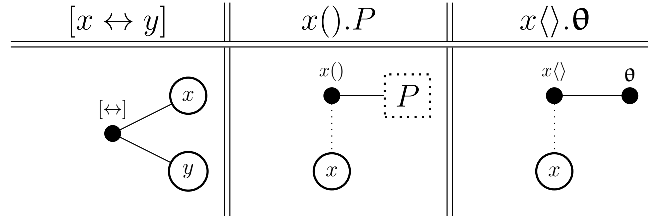
- $[x \leftrightarrow y]$. Forward the communication of channel x in y; it behaves like an identity channel.

- $x().P$. Suspension of the process P.

- $x\langle\rangle.\mathbf{\theta}$. Termination in communication.

It is important to point out that the processes considered by the πULL system are not the same as those in the π-calculus. For example, the replication is restricted only to the input. Despite this, the essence of the message passing is present in these new processes.

As a consequence of this new kind of processes, the π-calculus definitions and results may be changed or extended to fit the new concepts. The first notion that changes is the graphical representation for the processes. We consider three new rules.



For the πULL-calculus, the set of free names for a process is:

$$\begin{aligned}
fn(\mathbf{\theta}) &= \{\} \\
fn([x \leftrightarrow y]) &= \{x, y\} \\
fn(P \mid Q) &= fn(P) \cup fn(Q) \\
fn(\nu x.P) &= fn(P) \setminus \{x\} \\
fn(x\langle y\rangle.P) &= \{x, y\} \cup fn(P) \\
fn(x(y).P) &= (\{x\} \cup fn(P)) \setminus \{y\} \\
fn(x().P) &= \{x\} \cup fn(P) \\
fn(!x(y).P) &= (\{x\} \cup fn(P)) \setminus \{y\} \\
fn(x\langle\rangle.\mathbf{\theta}) &= \{x\}
\end{aligned}$$

The notions of fresh name, substitution, and change of bound names, extend to the new grammar naturally. Hence, the definition of α-equivalent processes does not change significantly. For example, we define the substitution as follows.

$$
\begin{aligned}
x\{z/w\} &= \begin{cases} z & x = w \\ x & \text{otherwise} \end{cases} \\
\mathbf{\theta}\{z/w\} &= \mathbf{\theta} \\
([x \leftrightarrow y])\{z/w\} &= [x\{z/w\} \leftrightarrow y\{z/w\}] \\
(P \mid Q)\{z/w\} &= (P\{z/w\} \mid Q\{z/w\}) \\
(\nu x.P)\{z/w\} &= \nu x.(P\{z/w\}) \\
(x\langle y\rangle.P)\{z/w\} &= x\{z/w\}\langle y\{z/w\}\rangle.(P\{z/w\}) \\
(x(y).P)\{z/w\} &= x\{z/w\}(y).(P\{z/w\}) \\
(x().P)\{z/w\} &= x\{z/w\}().(P\{z/w\}) \\
(!x(y).P)\{z/w\} &= !(x\{z/w\})(y).(P\{z/w\}) \\
(x\langle\rangle.\mathbf{\theta})\{z/w\} &= x\{z/w\}\langle\rangle.\mathbf{\theta}
\end{aligned}
$$

In this definition, we require some side conditions in the substitutions, for example $x \neq w$ and $x \neq z$ in $(\nu x.P)\{z/w\}$. These are the *no bound variable capture* from π-calculus.

In the case of the structural congruence and processes reduction, the equations/rules change, because congruences and reductions are obtained from the principal cuts [Wad12a, CPT12].

Definition 2.2. *Given two processes P, Q, we will say that they are structural congruent in πULL-calculus, if we can transform one into the other using the following equations (in either direction):*

$$
\begin{aligned}
P \mid Q &\equiv Q \mid P \\
P \mid \nu x.Q &\equiv \nu x.(P \mid Q), \text{ if } x \notin fn(P)
\end{aligned}
$$

Next, we define the reduction of terms ($\longrightarrow$):

$$\frac{}{\nu x.(P \mid [x \leftrightarrow y]) \longrightarrow P\{y/x\}}\ \textit{(fuse)} \qquad \frac{}{\nu x.(P \mid [y \leftrightarrow x]) \longrightarrow P\{y/x\}}\ \textit{(fuse)}$$

$$\frac{}{\nu x.([x \leftrightarrow y] \mid P) \longrightarrow P\{y/x\}}\ \textit{(fuse)} \qquad \frac{}{\nu x.([y \leftrightarrow x] \mid P) \longrightarrow P\{y/x\}}\ \textit{(fuse)}$$

$$\frac{}{\nu x.(x\langle\rangle.\mathbf{\theta} \mid x().P) \longrightarrow P}\ \textit{(1)} \qquad \frac{}{\nu x(x().P \mid x\langle\rangle.\mathbf{\theta}) \longrightarrow P}\ \textit{(1)}$$

$$\frac{}{\nu u.(!u(x).P \mid \nu y.u\langle y\rangle.Q) \longrightarrow \nu u.(!u(x).P \mid \nu y.(Q \mid P\{y/x\}))}\ \textit{(copy)}$$

$$\frac{}{\nu u.(!u(x).P \mid \nu y.u\langle y\rangle.Q) \longrightarrow \nu u.(!u(x).P \mid \nu y.(P\{y/x\} \mid Q))}\ \textit{(copy)}$$

$$\frac{}{\nu x.(x(z).P \mid \nu y.x\langle y\rangle.(Q \mid R)) \longrightarrow \nu y.(\nu x.(P\{y/z\} \mid R) \mid Q)}\ \textit{(M. Disjunction)}$$

$$\frac{}{\nu x.(\nu y.x\langle y\rangle(P \mid Q) \mid x(z).R) \longrightarrow \nu x.(Q \mid \nu y.(P \mid R\{y/z\}))}\ \textit{(M. Conjunction)}$$

Despite the rule *rep* was admissible in the π-calculus (Proposition 1.6) in the πULL-calculus, this rule is no longer admissible due to the lack of the equation for replication.

The next step in the construction is to explain the components of the system.

Definition 2.3. *πULL is a typing inference system that annotates sequents with terms and channel names to form typing judgments of the form:*

$$\Gamma; \Delta \vdash P :: \Lambda$$

Here Γ (resp. Δ, Λ) is the unrestricted (resp. linear) context of P. The contexts Γ, Δ and Λ consist of assignments of the form $x : A$ where x is a channel and A is a proposition/type. The dot $\cdot$ denotes the empty context.

A sequent can be read as follows: the process P offers the resources in Λ using the resources in Γ, Δ.

Additionally, we assume that the names for the unrestricted context are taken differently from the names in the linear context, in other words, there are two sets of names, one ordinary and other linear which are disjoint.

Now, we start the discussion about the system rules. We start presenting the cut rules. There are four different cut rules in the ULL system. The motivation for these rules is: there is a process that needs a resource of type A, and there is a process that consumes it. Then for the right cut rule, the premises should look like:

$$\Gamma; \Delta \vdash P :: \Lambda, x : A \qquad\qquad \Gamma; \Delta', x : A \vdash Q :: \Lambda'$$

Here the process P is capable of offering A along x, and Q requires A along x. Hence, the processes can be composed, but the communication must be private between them; and this implies that channel x should be private (restricted) for the composition. As a result, we obtain the following rule.

$$\frac{\Gamma; \Delta \vdash P :: \Lambda, x : A \qquad \Gamma; \Delta', x : A \vdash Q :: \Lambda'}{\Gamma; \Delta, \Delta' \vdash \nu x.(P \mid Q) :: \Lambda, \Lambda'}\ \text{(cutR)}$$

The left-cut rule is derived likewise. It is important to notice that, in the left-cut rule the cut proposition is the dual formula; this is consistent with the idea that the duality ensures input on one side and output on the other. We will not show the details for this case (and some others) here, but all the typing and cut reduction rules can be found in Appendix A.1.

The next cut rule corresponds with the WhyNot cut. Now the premises are:

$$\Gamma; x : A^{\perp} \vdash P :: \cdot \qquad\qquad \Gamma, u : A; \Delta \vdash Q :: \Lambda$$

This rule has a particular detail: the resource A that is being cut works in an unrestricted fashion for Q; hence it is necessary to provide the resource an indeterminate number of times. Fortunately, P can do this, due to its requirement of a process that works dual to A and its empty linear context. In πULL, this means a replicate input process. The rule obtained is:

$$\frac{\Gamma; x : A^{\perp} \vdash P :: \cdot \qquad \Gamma, u : A; \Delta \vdash Q :: \Lambda}{\Gamma; \Delta \vdash \nu u.(!u(x).P \mid Q) :: \Lambda}\ (\text{cut?})$$

Note that $!u(x).P$ replicates an input process, but this may be contrary to the intuition that P provides a service. The meaning is: process Q needs to provide its own name channel x over which the resource of type A is sent. This fulfills the requirement of a process of type A for Q while P can keep listening to new requests. A similar argument works to obtain the OfCourse cut rule.

Next, we present the identity rule. The idea behind this rule is: there is a need for a resource of type A in channel x whereas it is available a resource of the same type by channel y; then the resource can be forwarded by the channel y. This idea gives rise to the left and right identity rules.

$$\frac{}{\Gamma; x : A \vdash [x \leftrightarrow y] :: y : A}\ (\text{IdR})$$

Heuvel and Perez state the following about the validity of their rules:

> [vdHP20, p. 4] Caires, Pfenning, and Toninho showed that the validity of session type interpretations of linear logic propositions can be demonstrated by checking that cut reductions in typing inferences do correspond to reductions of processes, as well as by showing that the identity axiom of any type can be expanded to a larger process term with forwarding of a smaller type.

The aim of the present work is to exhibit the details for the mechanization/formal verification of the πULL system, following Caires et. al.'s [CPT12] presentation. We describe the rules to understand their nature and ideas, this would help us to understand better the rules and know how to implement it in Coq.

Also, we explain the principal cuts [CPT12]. The principal cuts show us that our reduction rules are well chosen. The idea is to study the cut rules, and study their behaviour with respect to other rules. We present in detail some cases, also we expect that this can guide anyone who wants to work on all cases.

We are going to start with the principal cut for the right identity rule. If we use the right-cut there are two possibilities: offering or consuming the channel being cut. In both cases, we obtain that the identity rule does correspond with the fuse reduction rule.

$$\dfrac{\dfrac{}{\Gamma; x : A \vdash [x \leftrightarrow y] :: y : A}\ (\text{IdR}) \qquad \Gamma; \Delta, y : A \vdash Q :: \Lambda}{\Gamma; x : A, \Delta \vdash \nu y.([x \leftrightarrow y] \mid Q) :: \Lambda}\ (\text{cutR})$$

This cut is eliminated as follows:

$$\Gamma; \Delta, x : A \vdash Q\{x/y\} :: \Lambda$$

Similar results are reached when combining the identity with the right or left cut rule. Now, we discuss the rules for type **1**. As Caires et. al. [CPT12] expose, from the linear logic rules for **1** there is nothing new expected or produced; then it is not necessary to output something or input anything when using these types. The only available action is to close the communication. This yields the following rules:

$$\dfrac{\Gamma; \Delta \vdash P :: \Lambda}{\Gamma; \Delta, x : \mathbf{1} \vdash x().P :: \Lambda}\ (\text{1L}) \qquad\qquad \dfrac{}{\Gamma; \cdot \vdash x\langle\rangle.\boldsymbol{\theta} :: x : \mathbf{1}}\ (\text{1R})$$

The associated cut reduction is the following:

$$\dfrac{\dfrac{}{\Gamma; \cdot \vdash x\langle\rangle.\boldsymbol{\theta} :: x : \mathbf{1}}\ (\mathbf{1}R) \qquad \dfrac{\Gamma; \Delta \vdash P :: \Lambda}{\Gamma; \Delta, x : \mathbf{1} \vdash x().P \vdash :: \Lambda}\ (\mathbf{1}L)}{\Gamma; \Delta \vdash \nu x.(x\langle\rangle.\boldsymbol{\theta} \mid x().P) :: \Lambda}\ (\text{CutR})$$

Which reduces to:

$$\Gamma; \Delta \vdash P :: \Lambda$$

Here is important to point out that there are no more interactions between the cut rules and the typing rules for **1**; all other cases lead to situations in which both premises of the cut are equal. For the bottom process, we can use similar ideas to obtain the corresponding rules and reductions.

The next group is the OfCourse rules. The right uses a similar idea to the one employed in the WhyNot cut, offering a replicate input process. For the left, we promote a channel from unrestricted to linear; thus, we require a change of names. The same arguments apply in the derivation of the WhyNot rules.

$$\frac{\Gamma, u : A; \Delta \vdash P :: \Lambda}{\Gamma; \Delta, x :!A \vdash P\{x/u\} :: \Lambda}\ (!\mathrm{L}) \qquad \frac{\Gamma; \cdot \vdash P :: y : A}{\Gamma; \cdot \vdash !x(y).P :: x :!A}\ (!R)$$

The cut reduction is:

$$\frac{\dfrac{\Gamma; \cdot \vdash P :: y : A}{\Gamma; \cdot \vdash !x(y).P :: x :!A}\ (!\mathrm{R}) \qquad \dfrac{\Gamma, u : A; \Delta \vdash Q :: \Lambda}{\Gamma; \Delta, x :!A \vdash Q\{x/u\} :: \Lambda}\ (!\mathrm{L})}{\Gamma; \Delta \vdash \nu x.(!x(y).P \mid Q\{x/u\}) :: \Lambda}\ (\mathrm{CutL})$$

Eliminating the cut yields:

$$\frac{\Gamma; \cdot \vdash P :: y : A \qquad \Gamma, u : A; \Delta \vdash Q :: \Lambda}{\Gamma; \Delta \vdash \nu u.(!u(y).P \mid Q)}\ (\mathrm{Cut!})$$

This does not correspond to a reduction, but it makes sense because the channels in processes P and Q are not interacting. Applying the same idea we can derive the WhyNot rules. Although, there is no cut for WhyNot rules due to the dual type in the right rule, which does not match with the left case.

In the case of the Copy rules, the intuition is this: P offers a resource of type A along channel x, and at the same time in an unrestricted way along channel u. Hence, it is possible to request a type A, we need to create a new channel each time.

$$\frac{\Gamma, u : A; \Delta \vdash P :: \Lambda, x : A^{\perp}}{\Gamma, u : A; \Delta \vdash \nu x.u\langle x\rangle.P :: \Lambda}\ (\mathrm{CopyR})$$

The cut instance for this rule is the following.

$$\frac{\Gamma; y : A^{\perp} \vdash P :: \cdot \qquad \dfrac{\Gamma, u : A; \Delta, x : A \vdash Q :: \Lambda}{\Gamma, u : A; \Delta \vdash \nu x.(u\langle x\rangle.Q) :: \Lambda}\ (\mathrm{CopyL})}{\Gamma; \Delta \vdash \nu u.(!u(y).P \mid \nu x.u\langle x\rangle.Q)) :: \Lambda}\ (\mathrm{Cut?})$$

Whose associate cut-elimination is:

$$\dfrac{\Gamma; y : A^{\perp} \vdash P :: \cdot \qquad \dfrac{\Gamma; x : A^{\perp} \vdash P\{x/y\} :: \cdot \qquad \Gamma, u : A; \Delta, x : A \vdash Q :: \Lambda}{\Gamma, u : A; \Delta \vdash \nu x.(Q \mid P\{x/y\}) :: \Lambda}\text{(CutL)}}{\Gamma; \Delta \vdash \nu u.(!u(y).P \mid \nu x.(Q \mid P\{x/y\}) :: \Lambda}\text{(Cut?)}$$

Remember that $A ⅋ B = A^{\perp} \multimap B$. Now, for multiplicative disjunction right rule, the situation is as follows: P needs a resource of type $A^{\perp}$ to produce a resource of type B; this translates into an input of type $A^{\perp}$ over a channel, and then to offer a B over the same channel. Note, that an input of type $A^{\perp}$ is an output of type A.

$$\dfrac{\Gamma; \Delta \vdash P :: \Lambda, x : B, y : A}{\Gamma; \Delta \vdash x(y).P :: \Lambda, x : A ⅋ B}(⅋R)$$

In the case of the left rule, a process P needs a resource of type A (which is an output of type $A^{\perp}$. On the other hand, a process Q needs a proposition of type B. As a result, it is necessary to expect a resource of type A first; then, we output it over a new channel, lets call it y. Finally, y needs to wait for a resource of type B.

$$\dfrac{\Gamma; \Delta, y : A \vdash P :: \Lambda \qquad \Gamma; \Delta', x : B \vdash Q :: \Lambda'}{\Gamma; \Delta, \Delta', x : A ⅋ B \vdash \nu y.x\langle y\rangle.(P \mid Q) :: \Lambda, \Lambda'}(⅋L)$$

Here the channel restriction (ν) in the left rule fulfills the commitment that the channel is fresh. The cut elimination follows:

$$\dfrac{\dfrac{\Gamma; \Delta_1 \vdash P :: \Lambda_1, x : B, z : A}{\Gamma; \Delta_1 \vdash x(z).P :: \Lambda_1, x : A ⅋ B}(⅋R) \qquad \dfrac{\Gamma; \Delta_2, y : A \vdash Q :: \Lambda_2 \qquad \Gamma; \Delta_3, x : B \vdash R :: \Lambda_3}{\Gamma; \Delta_2, \Delta_3, x : A ⅋ B \vdash \nu y.x\langle y\rangle.(Q \mid R) :: \Lambda_2, \Lambda_3}(⅋L)}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \nu x.(x(z).P \mid \nu y.x\langle y\rangle.(Q \mid R)) :: \Lambda_1, \Lambda_2, \Lambda_3}\text{(CutR)}$$

Whose reduction is:

$$\dfrac{\dfrac{\Gamma; \Delta_1 \vdash P\{y/z\} :: \Lambda_1, x : B, y : A \qquad \Gamma; \Delta_3, x : B \vdash R :: \Lambda_3}{\Gamma; \Delta_1, \Delta_3 \vdash \nu x.(P\{y/z\} \mid R) :: \Lambda_1, \Lambda_3, y : A}\text{(CutR)} \qquad \Gamma; \Delta_2, y : A \vdash Q :: \Lambda_2}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \nu y.(\nu x.(P\{y/z\} \mid R) \mid Q) :: \Lambda_1, \Lambda_2, \Lambda_3}\text{(CutR)}$$

Finally, to close the discussion, we present the rules for the multiplicative conjunction. The right rule has two processes P, Q; P provides a resource A over the channel y, and Q provides a resource B on channel x. Hence, to provide a resource of type $A \otimes B$ it is sufficient to output a resource A over a new channel y, then output a resource B on x.

$$\frac{\Gamma; \Delta \vdash P :: \Lambda, y : A \qquad \Gamma; \Delta' \vdash Q :: \Lambda', x : B}{\Gamma; \Delta, \Delta' \vdash \nu y.x\langle y\rangle.(P \mid Q) :: \Lambda, \Lambda', x : A \otimes B} (\otimes R)$$

For the left rule, the process P expects two resources: one of type A over a channel y and one of type B over channel x. Hence, it is possible to accept both resources over the same channel by first expecting the A resource, then waiting resource B.

$$\frac{\Gamma; \Delta, y : A, x : B \vdash P :: \Lambda}{\Gamma; \Delta, x : A \otimes B \vdash x(y).P :: \Lambda} (\otimes L)$$

The cut is the following.

$$\frac{\dfrac{\Gamma; \Delta_1 \vdash P :: \Lambda_1, y : A \qquad \Gamma; \Delta_2 \vdash Q :: \Lambda_2, x : B}{\Gamma; \Delta_1, \Delta_2 \vdash \nu y.x\langle y\rangle.(P \mid Q) :: \Lambda_1, \Lambda_2, x : A \otimes B} (\otimes R) \qquad \dfrac{\Gamma; \Delta_3, z : A, x : B \vdash R :: \Lambda_3}{\Gamma; \Delta_3, x : A \otimes B \vdash x(z).R :: \Lambda_3} (\otimes L)}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \nu x.(\nu y.x\langle y\rangle.(P \mid Q) \mid x(z).R) :: \Lambda_1, \Lambda_2, \Lambda_3} (\text{CutR})$$

And the reduction is:

$$\frac{\Gamma; \Delta_2 \vdash Q :: \Lambda_2, x : B \qquad \dfrac{\Gamma; \Delta_1 \vdash P :: \Lambda_1, y : A \qquad \Gamma; \Delta_3, y : A, x : B \vdash R\{y/z\} :: \Lambda_3}{\Gamma; \Delta_1, \Delta_3, x : B \vdash \nu y.(P \mid R\{y/z\}) :: \Lambda_1, \Lambda_3} (\text{CutR})}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \nu x.(Q \mid \nu y.(P \mid R\{y/z\})) :: \Lambda_1, \Lambda_2, \Lambda_3} (\text{CutR})$$

Remember that a summary of all the typing and cut reduction rules can be found in Appendix A.1.

2.2 Subject Reduction

We have reached one of the most important parts of this work: the proof of the Subject Reduction Theorem. We discussed the importance of this property: it means a system does not have deadlock or infinite wait errors.

Our Subject Reduction Theorem proof will be by induction on the typing derivation. To complete the proof, we will require some technical results to solve some cases. For example, most of the cases will not reduce. For other cases, we need to study the interaction between the πULL rules and the reduction rules. And, we will study the properties of the sequents to complete the proof.

Now, we will focus on these technical results. Most of them come from the work on Coq. We design them to make the verification straightforward.

Lemma 2.1. *The following processes do not reduce.*

- $[x \leftrightarrow y]$.
- $\nu x.u\langle x\rangle.P$.

- $x().P$.
- $x\langle\rangle.\mathbf{0}$.
- $x(y).P$.
- $\nu y.x\langle y\rangle.(P \mid Q)$.
- $!x(y).P$.

Proof. In all cases, there is no matching rule. □

The following lemmas tell how behave the substitution and the reduction. Their proofs require several applications of the α-equivalence, a common notion whose implementation in Coq is non trivial.

Lemma 2.2. *If $w \notin fn(S)$ and $S \longrightarrow T$. Then for all names b, $S\{w/b\} \longrightarrow T\{w/b\}$.*

Proof. The proof is by induction on the reduction rule:

- In the case of the rule $\nu x.(P \mid [x \leftrightarrow y]) \longrightarrow P\{y/x\}$ by the α-equivalence it is true that $w \neq x$, $b \neq x$, and as $w \notin fn(S)$ then $w \neq y$. Therefore, there are two possible cases:
  - ⋆ $b = y$. It is the case that $\nu x.(P \mid [x \leftrightarrow y])\{w/y\} = \nu x.(P\{w/y\} \mid [x \leftrightarrow w])$, on the other hand $P\{y/x\}\{w/y\} = P\{w/x\}$, as $y \notin fn(P)$; and $P\{w/y\} = P$; henceforth we can conclude this case.
  - ⋆ $b \neq y$. In this case $\nu x.(P \mid [x \leftrightarrow y])\{w/b\} = \nu x.(P\{w/b\} \mid [x \leftrightarrow y])$, additionally $P\{y/x\}\{w/b\} = P\{w/b\}\{y/x\}$ given the conditions over x, y, w, and b.

  All the other fusion rules are similar to the previous one.

- $\nu x(x\langle\rangle.\mathbf{0} \mid x().P) \longrightarrow P$. Using the α-equivalence it is the case that $w \neq x$ and $b \neq x$, therefore $\nu x.(x\langle\rangle.\mathbf{0} \mid x().P)\{w/b\} = \nu x.(x\langle\rangle.\mathbf{0} \mid x().P\{w/b\})$ whose reduction is $P\{w/b\}$. The other **1** rule is similar.

- In the case of $\nu.(!u(x).P \mid \nu y.u\langle y\rangle.Q) \longrightarrow \nu u.(!u(x).P \mid \nu y.(Q \mid P\{y/x\}))$, using the α-equivalence it is the case that $w \neq w$, $w \neq y$, $w \neq x$, $b \neq x$, $b \neq u$, and, $b \neq y$. Then it holds that $\nu u.(!u(x).P \mid \nu y.u\langle y\rangle.Q)\{w/z\} = \nu u.(!u(x).P\{w/z\} \mid \nu y.u\langle y\rangle.Q\{w/z\})$, whose reduction is $\nu u.(!u(x).P\{w/z\} \mid \nu y.(Q\{w/z\} \mid P\{w/z\}\{y/x\}))$.

  On the other side,

$\nu u.(!u(x).P \mid \nu y.(Q \mid P\{y/x\}))\{w/z\} = \nu u.(!u(x).P\{w/z\} \mid \nu y.(Q\{w/z\} \mid P\{y/x\}\{w/z\}))$.

Hence, using the conditions over the names it is true that $P\{y/x\}\{w/z\} = P\{w/z\}\{y/x\}$, and the conclusion follows.

- For multiplicative disjunction and conjunction we use the same idea of the previous case, analysing the conditions on the names that arise as consequence of the α-equivalence. □

Lemma 2.3. *If $w \notin fn(P)$ and $P \longrightarrow Q$, then $w \notin fn(Q)$.*

Proof. Similar to the proof of Lemma 2.2. □

We require the following lemmas to obtain the sequents in the cut cases. The first result is a case of identity cuts.

Lemma 2.4. *(left substitution) If $\Gamma; \Delta \vdash P :: \Lambda$, $z : B \in \Delta$, and $w \notin fn(\Gamma, \Delta, \Lambda, P)$ then, $\Gamma; \Delta' \vdash P\{w/z\} :: \Lambda$, here Δ' is the context in which $z : B$ is replaced by $w : B$.*

Proof. The proof is by induction on P's typing derivation.

- IdR. Then it holds $z = x$, therefore the conclusion is:

$$\Gamma; w : B \vdash [x \leftrightarrow y]\{w/x\} :: y : B$$

  And, as $y \neq x$, we conclude:

$$\Gamma; w : B \vdash [w \leftrightarrow y] :: y : B$$

  Which is an initial sequent.

- IdL. There are two cases: $z = x$, and $z = y$, which can be solved using the idea of the previous case.

- !R. The hypothesis does not match with this rule.

- !L. First, recall that $u \neq w$ since the linear names are disjoint from the ordinary ones. In this case, the typing has the form:

$$\frac{\Gamma, u : A; \Delta \vdash P :: \Lambda}{\Gamma; \Delta, x :!A \vdash P\{x/u\} :: \Lambda} \ (!L)$$

And there are two cases to consider: $x = z$ and $z : B \in \Delta$. In the first, we need the substitution $P\{x/u\}\{w/x\}$ which is equal to $P\{w/u\}$, and the typing required is:

$$\Gamma; \Delta_0, w :!A \vdash P\{w/u\} :: \Lambda$$

This sequent can be obtained by applying the !L rule. For the second case, if $\Delta = z : B, \Delta_0$; by the induction hypothesis we obtain the sequent:

$$\Gamma, u : A; \Delta_0, w : B \vdash P\{w/z\} :: \Lambda$$

because $x \neq w$ by hypothesis and $x \neq z$, by the !L rule we get:

$$\Gamma; \Delta_0, w : B, x :!A \vdash P\{w/z\}\{x/u\} :: \Lambda$$

Using the conditions on the names, $P\{w/z\}\{x/u\} = P\{x/u\}\{w/z\}$ can be derived and then, the result follows.

- ?R. In this case $\Delta = \Delta_0, z : B$ and applying the induction hypothesis to:

$$\Gamma, u : A; \Delta_0, z : B \vdash P :: \Lambda$$

We obtain

$$\Gamma, u : A; \Delta_0, w : B \vdash P\{w/z\} :: \Lambda$$

Taking into account that $x \neq u$, $x \neq w$, and $x \neq z$, then $P\{x/u\}\{w/z\} = P\{w/z\}\{x/u\}$. This last equation implies:

$$\Gamma; \Delta_0, w : B \vdash P\{x/u\}\{w/z\} :: \Lambda, x :?A$$

- ?L. In this case, it follows that $x = z$ and as $x \notin fn(P)$, hence $P\{w/x\} = P$ and we obtain:

$$\Gamma; w :?A \vdash !w(y).P :: \cdot$$

as required.

- $\parr L$. The hypothesis, in this case, looks like this:

$$\frac{\Gamma; \Delta_1, y : A \vdash P :: \Lambda_1 \qquad \Gamma; \Delta_2, x : B \vdash Q :: \Lambda_2}{\Gamma; \Delta_1, \Delta_2, x : A \parr B \vdash \nu y.x\langle y\rangle.(P \mid Q) :: \Lambda_1, \Lambda_2} (\parr L)$$

Then, there are three cases for z. The first case is when $z = x$, in which using the induction hypothesis in the right sequent we get:

$$\Gamma; \Delta_2, w : B \vdash Q\{w/x\} :: \Lambda_2$$

As contexts in the sequents are disjoint it holds that $P\{w/x\} = P$. Thus, by α-equivalence $w \neq y$ and:

$$\Gamma; \Delta_1, \Delta_2, w : A \parr B \vdash \nu y.w\langle y\rangle.(P \mid Q\{w/x\}) :: \Lambda_1, \Lambda_2$$

The second case is when $\Delta_2 = \bar{\Delta}_2, z : \bar{B}$, then applying the induction hypothesis we get:

$$\Gamma; \bar{\Delta}_2, w : \bar{B}, x : B \vdash Q\{w/z\} :: \Lambda_2$$

Note the following, by α-equivalence $y \neq w$, and given that the contexts (in the sequents) are disjoint $z \notin fn(P)$, $P\{w/z\} = P$, and $x \neq z$. Therefore, we can get the following sequent:

$$\Gamma; \Delta_1, \bar{\Delta}_2, w : \bar{B}, x : A \parr B \vdash \nu y.x\langle y\rangle.(P \mid Q\{w/z\}) :: \Lambda_1, \Lambda_2$$

The third case is when $\Delta_1 = \bar{\Delta}_1, z : \bar{B}$, and is similar to the previous one.

- $\parr R$. In this case, $\Delta = \bar{\Delta}, z : \bar{B}$ and by the induction hypothesis:

  $$\Gamma; \bar{\Delta}, w : \bar{B} \vdash P\{w/z\} :: \Lambda, x : B, y : A$$

  By α-equivalence $w \neq y$ and by the hypothesis $w \neq x$, hence:

  $$\Gamma; \bar{\Delta}, w : \bar{B} \vdash x(y).(P\{w/z\}) :: \Lambda, x : A \parr B$$

- $\otimes L$, and $\otimes R$. They are similar to the two previous cases.
- $\perp R$ or $\mathbf{1}L$. In this case $\Delta = \bar{\Delta}, z : B$ and by the induction hypothesis:

  $$\Gamma; \bar{\Delta}, w : B \vdash P\{w/z\} :: \Lambda$$

  Which leads to:

  $$\Gamma; \bar{\Delta}, w : B \vdash x().(P\{w/z\}) :: \Lambda, x : \perp$$

  Applying the rule $\perp R$ or $\mathbf{1}L$.

- $\perp L$. In this case $z = x$ and it follows by the same rule.
- $\mathbf{1}R$. The rules do not match.
- CopyL or CopyR. It is necessary that $\Delta = \bar{\Delta}, z : B$ and using the α-equivalence, $x \neq w$. So by the induction hypothesis:

$$\Gamma, u : A; \bar{\Delta}, w : B \vdash P\{w/z\} :: \Lambda, x : A^{\perp}$$

And the result can be derived using the case rule.

- Cut? or Cut!. It follows that $\Delta = \bar{\Delta}, z : B$ and by the α-equivalence $w \neq x$ and $w \neq u$. Thus:

$$\Gamma, u : A; \bar{\Delta}, w : B \vdash Q\{w/z\} :: \Lambda$$

So by an application of the same rule, it follows the required sequent.

- CutR or CutL. There are two cases in each rule. The first case is when $\Delta_1 = \bar{\Delta_1}, z : B$ in which by the induction hypothesis we derive:

$$\Gamma; \bar{\Delta_1}, w : B, x : A \vdash P\{w/z\} :: \Lambda$$

By α-equivalence, $w \neq x$; since the contexts are disjoint $Q\{w/z\} = Q$, and we finish applying the same cut rule under consideration. The second case is when $\Delta_2 = \bar{\Delta_2}, z : B$ and it is similar.

In all cases the result is obtained. Using induction on the typing rules, it follows the lemma. □

The following lemma and its proof is similar to the previous one.

Lemma 2.5. *(right substitution) If $\Gamma; \Delta \vdash P :: \Lambda$, $z : B \in \Lambda$, and $w \notin fn(\Gamma, \Delta, \Lambda, P)$ then, $\Gamma; \Delta \vdash P\{w/z\} :: \Lambda'$, here Λ' is the context in which $z : B$ is replaced by $w : B$.*

Proof. Similar to the proof of Lemma 2.4. □

The following lemmas provide transference principals, which allow us to move one assignment from one to the other side of the turnstile

Lemma 2.6. *(right to left transference) If $\Gamma; \Delta \vdash P :: \Lambda$ and $w : B \in \Lambda$ then, $\Gamma; \Delta, w : B^{\perp} \vdash P :: \Lambda'$ being $\Lambda' = \Lambda \setminus \{w : B\}$.*

Proof. By induction on the typing derivation.

- The derivation cannot be obtained by applying the rules IdL, ?L or $\perp$L.
- If the last rule is CopyL it follows that $w : B \in \Lambda$, then using the induction hypothesis we get $\Gamma, u : A; \Delta, w : B^\perp \vdash P :: \Lambda'$, and the required conclusion follows using the same rule (CopyL).
- In the case of CopyR, Cut!, Cut?, **1**L, $\otimes$L, or !L the proof is similar to the previous one.
- For the rules CutR, CutL, or $⅋$L, we apply the induction hypothesis depending on which context lies the assignment $w : B$.
- If the last rule is IdR, we require to derive $\Gamma; x : A, w : A^\perp \vdash [x \leftrightarrow w] :: \cdot$, but this can be obtained using IdL.
- If the last rule is **1**R then $w : B = x : \mathbf{1}$ and we require $\Gamma; x :\perp \vdash x\langle\rangle.\mathbf{0} :: \cdot$; this follows using the $\perp$L rule.
- In the case of rule $\perp$R, there are two cases: in the first $x : B \in \Lambda$, and the conclusion is obtained by the induction hypothesis. In the second, $x : B = x :\perp$ and the required sequent is $\Gamma; \Delta, x : \mathbf{1} \vdash x().P :: \Lambda$, which can be typed using the **1**L rule.
- For the $?R$ and $⅋$R rules, the proof is similar to the previous one.
- If the last applied rule is $\otimes$R, there are three cases. If $w : B \in \Lambda$ or $w : B \in \Lambda'$, use the induction hypothesis. To prove in the case when $w : B = x : A \otimes B$, we use the $⅋$ L rule and induction.
- Finally, in the case of $!R$ rule the desired conclusion is that $\Gamma; x :?A^\perp \vdash\ !x(y).P :: \cdot$, additionally by the induction hypothesis $\Gamma; y : A^\perp \vdash P :: \cdot$, hence the conclusion is obtained applying the ?L rule. □

Lemma 2.7. *(left to right) If* $\Gamma; \Delta \vdash P :: \Lambda$ *and* $w : C \in \Delta$ *then,* $\Gamma; \Delta' \vdash P :: \Lambda, w : C^\perp$ *being* $\Delta' = \Delta \setminus \{w : C\}$.

Proof. Similar to Lemma 2.6. □

The next result helps to identify invalid sequents involving forward processes.

Lemma 2.8. *The following holds:*

- *For all contexts* Γ, Δ, *and* Λ, *free names* x, y, *and proposition* A *there is no derivation for* $x : A, \Gamma; \Delta \vdash [x \leftrightarrow y] :: \Lambda$.

- *For all contexts* Γ, Δ, *and* Λ, *free names* x, y, *and proposition* A *there is no derivation for* $x : A, \Gamma; \Delta \vdash [y \leftrightarrow x] :: \Lambda$.

- *If* $S = [x \leftrightarrow y]\{u/w\}$, $u \in fn(S)$, *and* $u : A \in \Gamma$ *then there is no derivation for* $\Gamma; \Delta \vdash S :: \Lambda$.

- *If* $S = [y \leftrightarrow x]\{u/w\}$, $u \in fn(S)$, *and* $u : A \in \Gamma$ *then there is no derivation for* $\Gamma; \Delta \vdash S :: \Lambda$.

Proof. For the first two, the proof is by induction on the typing derivation. We can have derivations using IdR or IdL rules, but in these cases channel x would have an ordinary and linear behavior, which is contradictory. The !L and ?R rules match the induction step, but they produce a contradiction on the premises sequents. So there is no possible derivation by induction.

For the third and fourth statements as $u \in fn(S)$, there are two possibilities: $x = w$ or $y = w$ .Both cases can be solved using the first two results. □

The next result is similar to the previous one but for the termination process.

Lemma 2.9. *The following are true:*

- *For all contexts* Γ, Δ, *and* Λ, *free name* x, *and proposition* A *there is no derivation for* $x : A, \Gamma; \Delta \vdash x\langle\rangle.\boldsymbol{\theta} :: \Lambda$.

- *If* $S = x\langle\rangle.\boldsymbol{\theta}\{u/w\}$, $u \in fn(S)$, *and* $u : A \in \Gamma$ *then there is no derivation for* $\Gamma; \Delta \vdash S :: \Lambda$.

Proof. Similar to the previous lemma. □

One feature of the ordinary context is that it has the weakening property.

Lemma 2.10. *(Ordinary Weakening) Suppose that* $\Gamma; \Delta \vdash P :: \Lambda$ *and* w *is an ordinary name that does not appear in all the derivation of the sequent then,* $\Gamma, w : B; \Delta \vdash P :: \Lambda$.

Proof. Induction on the height of the typing of $\Gamma; \Delta \vdash P :: \Lambda$:

- Rules IdR and IdL: the sequent has the form $\Gamma; x : A \vdash [x \leftrightarrow y] :: y : A$, and by the hypothesis over w, $w \neq x$ and $w \neq y$. Thus, the sequent $\Gamma, w : B; x : A \vdash [x \leftrightarrow y] :: y : A$ is initial.

- Rules !L and ?R: as w does not appear in all the derivation of the sequent, it is true that $w \neq u$ and then by induction hypothesis $\Gamma, w : B, u : A; \Delta \vdash P :: \Lambda$ (!L), then the conclusion follows applying the same rule.

- In the other cases, apply the induction hypothesis and the fact that w is ordinary. □

Now, we show some processes that cannot be typed in the system.

Lemma 2.11. *For all contexts* Γ*,* Δ*,* Λ*, and processes* P*,* Q*; there is no derivation for* $\Gamma; \Delta \vdash P \mid Q :: \Lambda$*.*

Proof. Induction on the derivation. □

Lemma 2.12. *For all contexts* Γ*,* Δ*,* Λ*, ordinary name* u *and proposition* A*, there is no derivation for* $\Gamma, u : A; \Delta \vdash u(y).P :: \Lambda$*.*

Proof. Induction on the derivation of the sequent. □

Lemma 2.13. *If* $\Gamma, u : A; \Delta \vdash \nu y.u\langle y\rangle.P :: \Lambda$ *is neither derived using the !L nor ?R rules, then it is derived using the CopyR or CopyL rule.*

Proof. The proof is by induction on the typing derivation:

- CopyR or CopyL rules: it is the conclusion.
- !L or ?R rules: it is not possible by hypothesis.
- $\parr$L or $\otimes$R rules: then u should have an ordinary and a linear behavior, which is a contradiction. □

Now, we can show that the πULL typing system preserves the typing under structural congruence.

Theorem 2.1. *If* $\Gamma; \Delta \vdash P :: \Lambda$ *and* $P \equiv Q$*, then* $\Gamma; \Delta \vdash Q :: \Lambda$*.*

Proof. The proof uses induction on the hypothesis $P \equiv Q$. But, in all cases there is no typing for P. □

Finally, we are able to present the Subject Reduction Theorem.

Theorem 2.2. *If* $\Gamma; \Delta \vdash P :: \Lambda$ *and* $P \longrightarrow Q$*, then* $\Gamma; \Delta \vdash Q :: \Lambda$*.*

Proof. The proof is by induction on the typing derivation:

- Using Lemma 2.1, the typing rule cannot be IdL, IdR, CopyL, CopyR, $\mathbf{1}L$, $\mathbf{1}R$, $\perp L$, $\perp R$, $\otimes L$, $\otimes R$, $\parr L$, $\parr R$, $!R$, or $?L$.

- For the rules $?R$ and $!L$, the next argument solves these cases. Here we consider the $!L$ rule. First, by the definition of the rule, it holds that $x \notin fn(P)$.

  By hypothesis, $P\{x/u\} \longrightarrow Q$ and it is clear that $u \notin fn(P)$; hence using Lemma 2.2, it follows $P\{x/u\}\{u/x\} \longrightarrow Q\{u/x\}$ and $P = P\{x/u\}\{u/x\}$. Using the induction hypothesis, $\Gamma, u : A; \Delta \vdash Q\{u/x\} :: \Lambda$ and applying the rule $!L$, it follows that $\Gamma; \Delta, x :!A \vdash Q\{u/x\}\{x/u\} :: \Lambda$.

  As $Q = Q\{u/x\}\{x/u\}$, we conclude the result.

- For the Cut! rule the hypothesis is $\nu u.(!u(x).P \mid R) \longrightarrow Q$ and there are multiple cases:

  ⋆ $R = [u \leftrightarrow y]$. It will be the case that $\Gamma, u : A; \Delta \vdash [u \leftrightarrow y] :: \Lambda$, by Lemma 2.8 this cannot happen.

  ⋆ $R = [y \leftrightarrow u]$. It is equal to the previous case.

  ⋆ $R = \nu y.u\langle y\rangle.S$ and the reduction is $\nu u.(!u(x).P \mid \nu y.(S \mid P\{y/x\}))$. By hypothesis, it is true that $\Gamma, u : A; \Delta \vdash \nu y.u\langle y\rangle.S :: \Lambda$, and we can obtain the last sequent using:

    * An application of the rules !L or ?R. Suppose that we applied the rule !L, there is a process S' such that $S'\{x/u_0\} = S$, and channels u_0, x such that:
      1. Are different from y by α-equivalence.
      2. Are different from u, because if $u = x$ then there would be a channel with double behavior. And, if $u = u_0$, then the application of the !L rule yields $\Gamma; \Delta \vdash \nu y.u\langle y\rangle.S :: \Lambda$ which is not the expected sequent.

      We obtain the typing:

$$\frac{\Gamma, u : A, u_0 : B; \Delta_0 \vdash (\nu y.u\langle y\rangle.S') :: \Lambda}{\Gamma, u : A; \Delta_0, x :!B \vdash (\nu y.u\langle y\rangle.S')\{x/u_0\} :: \Lambda} \text{(!L)}$$

      Then there are two possibilities for the upper sequent:

      · We derive it applying the rules !L or ?R. It is possible to apply the same argument to invert the sequent and obtain a similar one. If there are more than two consecutive applications of these rules, iterate the argument, as Δ and Λ are finite the process must stop and fall in the next case.

      · It is not obtained from the application of the rules !L nor ?R. Then this case is reduced to the others cases of rules.

From now, we will call this strategy for solving cases, as the cyclic argument.

* An application of the rules $⅋$L, or $\otimes$R. There is a contradiction on the behavior (linear and ordinary) of channel u, which is the channel of the current rule. For example in the $⅋$L:

$$\Gamma, u : A⅋B; \Delta, u : A⅋B \vdash \nu y.u\langle y\rangle.S :: \Lambda$$

The channel u behaves linear and ordinary, which is not possible.

* An application of the CopyR rule. It follows that $\Gamma, u : A; \Delta \vdash S :: \Lambda, y : A^\perp$. Using Lemmas 2.10, 2.6, and 2.4 it holds $\Gamma, u : A; y : A^\perp \vdash P\{y/x\} :: \cdot$, hence:

$$\dfrac{\Gamma; \cdot \vdash P :: x : A \qquad \dfrac{\Gamma, u : A; \Delta \vdash S :: \Lambda, y : A^\perp \qquad \Gamma, u : A; y : A^\perp \vdash P\{y/x\} :: \cdot}{\Gamma, u : A; \Delta \vdash \nu y.(S \mid P\{y/x\}) :: \Lambda}\,(\text{CutR})}{\Gamma; \Delta \vdash \nu u.(!u(x).P \mid \nu y.(S \mid P\{y/x\})) :: \Lambda}\,(\text{Cut!})$$

* An application of the CopyL rule. Then $\Gamma, u : A; \Delta, y : A \vdash S :: \Lambda$. By Lemmas 2.10, 2.6, and 2.4 we have $\Gamma, u : A; y : A^\perp \vdash P\{y/x\} :: \cdot$, hence:

$$\dfrac{\Gamma; \cdot \vdash P :: x : A \qquad \dfrac{\Gamma, u : A; \Delta, y : A \vdash S :: \Lambda \qquad \Gamma, u : A; y : A^\perp \vdash P\{y/x\} :: \cdot}{\Gamma, u : A; \Delta \vdash \nu y.(S \mid P\{y/x\}) :: \Lambda}\,(\text{CutL})}{\Gamma; \Delta \vdash \nu u.(!u(x).P \mid \nu y.(S \mid P\{y/x\})) :: \Lambda}\,(\text{Cut!})$$

⋆ $R = \nu y.u\langle y\rangle.S$ and the reduction is $\nu u.(!u(x).P \mid \nu y.(P\{y/x\} \mid S))$. By hypothesis we have $\Gamma, u : A; \Delta \vdash \nu y.u\langle y\rangle.S :: \Lambda$, and we can derive this sequent by:

* An application of the rules !L or ?R. We use the cyclic argument.

* An application of the rules $⅋$L, or $\otimes$R. There is a contradiction on the behavior (linear and ordinary) of channel u, which is the channel of the current rule. For example in the $⅋$L:

$$\Gamma, u : A⅋B; \Delta, u : A⅋B \vdash \nu y.u\langle y\rangle.S :: \Lambda$$

The channel u behaves linear and ordinary, which is not possible.

* An application of the CopyR rule. It follows that $\Gamma, u : A; \Delta \vdash S :: \Lambda, y : A^\perp$. Using Lemmas 2.10, 2.6, and 2.4 it holds $\Gamma, u : A; \Delta, y : A \vdash S :: \Lambda$ and $\Gamma, u : A; \cdot \vdash P\{y/x\} :: y : A$, hence:

$$\dfrac{\Gamma; \cdot \vdash P :: x : A \qquad \dfrac{\Gamma, u : A; \cdot \vdash P\{y/x\} :: y : A \qquad \Gamma, u : A; \Delta, y : A \vdash S :: \Lambda}{\Gamma, u : A; \Delta \vdash \nu y.(P\{y/x\} \mid S) :: \Lambda}\text{(CutR)}}{\Gamma; \Delta \vdash \nu u.(!u(x).P \mid \nu y.(P\{y/x\} \mid S)) :: \Lambda}\text{(Cut!)}$$

- ∗ An application of the CopyL rule. Then $\Gamma, u : A; \Delta, y : A \vdash S :: \Lambda$. By Lemmas 2.5 and 2.10 it is true that $\Gamma, u : A; \cdot \vdash P\{y/x\} :: y : A$, hence:

$$\dfrac{\Gamma; \cdot \vdash P :: x : A \qquad \dfrac{\Gamma, u : A; \cdot \vdash P\{y/x\} :: y : A \qquad \Gamma, u : A; \Delta, y : A \vdash S :: \Lambda}{\Gamma, u : A; \Delta \vdash \nu y.(P\{y/x\} \mid S) :: \Lambda}\text{(CutR)}}{\Gamma; \Delta \vdash \nu u.(!u(x).P \mid \nu y.(P\{y/x\} \mid S)) :: \Lambda}\text{(Cut!)}$$

- Now, for the CutR rule the hypothesis is $\nu x.(P \mid R) \longrightarrow Q$ and there are several cases depending on the reduction case:

  - ⋆ The rule is $\nu x.(P \mid [x \leftrightarrow y]) \longrightarrow P\{y/x\}$. So, $\Gamma; \Delta \vdash P :: \Lambda, x : A$ and $\Gamma; \Delta', x : A \vdash [x \leftrightarrow y] :: \Lambda'$ and there are cases depending on the typing of the last sequent:

    - ∗ IdR rule. The desire conclusion is $\Gamma; \Delta \vdash P\{y/x\} :: \Lambda, y : A$. But this is consequence of Lemma 2.5.
    - ∗ IdL rule. The objective is to type $\Gamma; \Delta, y : A^{\perp} \vdash P\{y/x\} :: \Lambda$. But we can derive it applying Lemmas 2.4 and 2.6.
    - ∗ By the rule !L or ?R. It is the case that:

$$\dfrac{\Gamma, u : A; \Delta'_0 \vdash [x \leftrightarrow y]\{u/w\} :: \Lambda'_0}{\Gamma; \Delta', x : A \vdash [x \leftrightarrow y] :: \Lambda'}\text{(!L)}$$

      with $u \in fn([x \leftrightarrow y]\{u/w\})$. Therefore, Lemma 2.8 states that there is a contradiction.

  - ⋆ The rule is $\nu x.(P \mid [y \leftrightarrow x]) \longrightarrow P\{y/x\}$. So, $\Gamma; \Delta \vdash P :: \Lambda, x : A$ and $\Gamma; \Delta', x : A \vdash [y \leftrightarrow x] :: \Lambda'$ and there are cases depending on the typing of the last sequent:

    - ∗ IdR rule. This would imply that $x = y$, but this is a contradiction.
    - ∗ IdL rule. This implies that $\Delta' = \{y : A^{\perp}\}$ and $\Lambda' = \emptyset$, the goal is to derive $\Gamma; \Delta, y : A^{\perp} \vdash P\{y/x\} :: \Lambda$ which we can derive using Lemmas 2.4 and 2.6.
    - ∗ By the rule !L or ?R. As previous this case satisfies the conditions to apply Lemma 2.8.

  - ⋆ The rule is $\nu x.([y \leftrightarrow x] \mid Q) \longrightarrow Q\{y/x\}$. Hence, $\Gamma; \Delta \vdash [y \leftrightarrow x] :: \Lambda, x : A$ and $\Gamma; \Delta', x : A \vdash Q :: \Lambda'$; there are cases depending on the typing of the last sequent:

  * IdR rule. This implies that $\Delta = \{y : A\}$ and $\Lambda = \emptyset$, the goal is to derive $\Gamma; \Delta', y : A \vdash Q\{y/x\} :: \Lambda'$ which we derive using Lemma 2.4.
  * IdL rule. It is impossible since the right side is not empty.
  * By the rule !L or ?R. As in the previous cases it satisfies the conditions to apply Lemma 2.8.

- ⋆ The rule is $\nu x.([x \leftrightarrow y] \mid Q) \longrightarrow Q\{y/x\}$. Thus, $\Gamma; \Delta \vdash [x \leftrightarrow y] :: \Lambda, x : A$ and $\Gamma; \Delta', x : A \vdash Q :: \Lambda'$; there are cases depending on the typing of the last sequent:
  * IdR rule. It does not match the typing.
  * IdL rule. It does not match the typing, the right side is not empty.
  * Rules !L or ?R. As previous this case satisfies the conditions to apply Lemma 2.8.

- ⋆ The rule is $\nu x.(x\langle\rangle.\mathbf{0} \mid x().Q) \longrightarrow Q$. In this case is true that $\Gamma; \Delta \vdash x\langle\rangle.\mathbf{0} :: \Lambda, x : A$ and $\Gamma; \Delta', x : A \vdash x().Q :: \Lambda'$; there are cases depending on the typing of the first sequent:
  * Rules !L or ?R. This case satisfies the conditions to apply Lemma 2.9.
  * Rule $\perp L$. There is a contradiction, the right side is not empty.
  * Rule $\mathbf{1}R$. Then $\Delta = \emptyset$, and $\Lambda = \emptyset$. The goal is $\Gamma; \Delta' \vdash Q :: \Lambda'$, and it is necessary to consider the typing for $\Gamma; \Delta', x : A \vdash x().Q :: \Lambda'$:
    · Rule $\mathbf{1}L$. The goal is the premise in the rule.
    · Rule $\perp R$. In this case the sequent has the form $\Gamma; \Delta', x : A \vdash x().Q :: \Lambda', x :\perp$, but the linear contexts are not disjoint, so it is contradictory.
    · Rules $!L$ or $?R$. We use the cyclic argument.

- ⋆ The rule is $\nu x.(x().P \mid x\langle\rangle.\mathbf{0}) \longrightarrow P$. The proof is similar to the previous case.
- ⋆ The rule is $\nu u.(!u(x).P \mid \nu y.u\langle y\rangle.Q) \longrightarrow \nu u.(!u(x).P \mid \nu y.(Q \mid P\{y/x\}))$. It is similar to the proof of the Cut! rule.
- ⋆ The rule is $\nu u.(!u(x).P \mid \nu y.u\langle y\rangle.Q) \longrightarrow \nu u.(!u(x).P \mid \nu y.(P\{y/x\} \mid Q))$. It is similar to the proof of the Cut? rule.
- ⋆ The rule is $\nu x.(x(z).P \mid \nu y.x\langle y\rangle.(Q \mid R)) \longrightarrow \nu y.(\nu x.(P\{y/z\} \mid R) \mid Q)$ in this case it is true that $\Gamma; \Delta \vdash x(z).P :: \Lambda, x : A$ and

$\Gamma; \Delta', x : A \vdash \nu y.x\langle y\rangle.(Q \mid R) :: \Delta'$, and there are cases depending on the second sequent:

- ∗ Rules CopyR or CopyL. It holds $\Gamma; \Delta', x : A, y : B \vdash Q \mid R :: \Delta'$ for some assignment $y : B$, but this contradicts Lemma 2.11.
- ∗ Rule ?R. We use the cyclic argument.
- ∗ Rule !L. As in the cyclic argument suppose that this is the first application of the rules !L or ?R. Then there are two cases. The first is that, x is the same name used by the rule, so it should be the case that $\Gamma, u : A; \Delta' \vdash \nu y.u\langle y\rangle(Q_0 \mid R_0) :: \Delta'$ which by Lemma 2.13 can only be concluded using the CopyL or CopyR rules, but this contradicts Lemma 2.11. Thus, the case reduces to the cyclic argument.
- ∗ Rule $\otimes$R. It is the case that $A = C \otimes D$ and $x : C \otimes D \in \Delta'$, this is contradictory due to the behavior of x.
- ∗ Rule ⅋L. In this case we obtain $A = C ⅋ D$, $\Gamma; \Delta_0, y : C \vdash Q :: \Lambda_0$ and $\Gamma; \Delta_1, x : D \vdash R :: \Lambda_1$ with $\Delta' = \Delta_0, \Delta_1$ and $\Lambda' = \Lambda_0, \Lambda_1$. Then, it is necessary to know how the sequent $\Gamma; \Delta \vdash x(z).P :: \Lambda, x : A$ were derived:
  - · Rule !L. We use the cyclic argument.
  - · Rule ?R. There are two possibilities. In the first x is the same name of the rule, but this implies that for some P_0, $\Gamma, u : A'; \Delta \vdash u(z).P_0 :: \Lambda$ which contradicts Lemma 2.12. Hence, the case reduces to the cyclic argument.
  - · Rule $\otimes$L This would imply that $\Gamma; \Delta_2, x : A' \otimes B' \vdash x(z).P :: \Lambda, x : C ⅋ D$ which is contradictory due to the double behavior in x.
  - · Rule ⅋R. Then $\Gamma; \Delta \vdash P :: \Lambda, x : D, z : C$ using Lemmas 2.4 and 2.6 we can get $\Gamma; \Delta, y : C^{\perp} \vdash P\{y/z\} :: \Lambda, x : D$ and the conclusion is derived as follows:

$$\dfrac{\dfrac{\Gamma; \Delta_0, y : C \vdash Q :: \Lambda_0 \qquad \Gamma; \Delta, y : C^{\perp} \vdash P\{y/z\} :: \Lambda, x : D}{\Gamma; \Delta, \Delta_0 \vdash \nu y.(Q \mid P\{y/z\}) :: \Lambda, \Lambda_0, x : D}\,(\text{CutL}) \qquad \Gamma; \Delta_1, x : D \vdash R :: \Lambda_1}{\Gamma; \Delta, \Delta_0, \Delta_1 \vdash \nu x.(\nu y.(Q \mid P\{y/z\}) \mid R) :: \Lambda, \Lambda_0, \Lambda_1}\,(\text{CutR})$$

- ⋆ The rule is $\nu x.(\nu y.x\langle y\rangle(P \mid Q) \mid x(z).R) \longrightarrow \nu x.(Q \mid \nu y.(P \mid R\{y/z\}))$ then it holds $\Gamma; \Delta \vdash \nu y.x\langle y\rangle(P \mid Q) :: \Lambda, x : A$ and $\Gamma; \Delta', x : A \vdash x(z).R :: \Lambda'$. Hence, there are cases depending on the first sequent:
  - ∗ Rules CopyR, CopyL, !L or ?R. It is similar to the previous rule.

  * Rule $\parr$L. Then it should be the case that $x : C \parr D \in \Delta$ for some propositions C, D, then x has a double behavior which is a contradiction.
  * Rule $\multimap$L. It is similar to the $\parr$L case.
  * Rule $\otimes$R. In this case $A = C \otimes D$ for some propositions C, and D; $\Gamma; \Delta_1 \vdash P :: \Lambda_1, y : C$, $\Gamma; \Delta_2 \vdash Q :: \Lambda_2, x : D$ with $\Delta = \Delta_1 \cup \Delta_2$ and $\Lambda = \Lambda_1 \cup \Lambda_2$. Now we need to know how $\Gamma; \Delta', x : A \vdash x(z).R :: \Lambda'$ were derived:
    · Rules !L or ?R. We use the cyclic argument.
    · Rule $\parr$R. It is the case that $\Gamma; \Delta', x : A \vdash x(z).R :: \Lambda'_0, x : B$, which is contradictory due to the double behavior of x.
    · Rule $\multimap$R. It is similar to the previous item.
    · Rule $\otimes$L. It is true that $\Gamma; \Delta'_0, x : C, z : D \vdash R :: \Lambda'$, using Lemma 2.4 we derive $\Gamma; \Delta'_0, x : C, y : D \vdash R\{y/z\} :: \Lambda'$, and the conclusion follows using two times the rule CutR.

- The CutL rule is similar to the previous case. □

In this section, we presented our version of the πULL calculus, and using the notion of principal cuts, we validated that the chosen reduction rules are coherent and make sense within the calculus. Furthermore, we completed the proof of the Subject Reduction Theorem, including the technical results required in the proof. The proof and its design are one of the major contributions of the present work.

Our next step will be the introduction of ideas for terms representation within computer machines. We need a representation to implement the verification of the Subject Reduction Theorem in Coq.

2.3 Bounded names and its mechanization

As we discussed in the Introduction, a good representation of definitions and concepts is fundamental when writing machine-checked proofs.

In systems like λ or π calculus, one keystone detail of the implementation is the representation of names (variables). This problem has been studied extensively, and there are different approaches [Cha12, de 72]. Following the discussion given by de Bruijn [de 72, p. 1], we consider the following three criteria for a good notation:

- easy to write and easy to read for the human reader;

- easy to handle in metalingual discussion;

- easy for the computer and the computer programmer.

There are three canonical ways to represent names within a computer machine: strings, De Bruijn notation, and Locally Nameless Representation (LN). Now, we give a brief discussion of them regarding their advantages and disadvantages. Also we present the LN representation in detail, which is the notation preferred in this work. As de Bruijn highlights [de 72, p. 383], presenting the examples using planar trees facilitates the discussion.

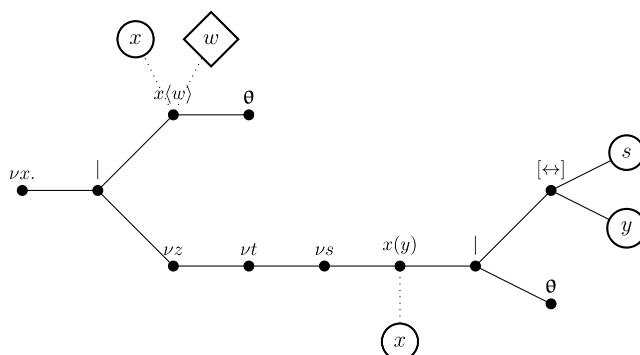
The naive representation for names uses strings of characters, as usual. We are used to this representation which is faithful to the mathematical definitions. The major drawback with this representation comes with the bounded names. To obtain a good representation of the terms, it is necessary to consider the α-equivalence as an equivalence relation over them.

This equivalence relation produces a technical difficulty: each function or relation, defined for the terms, must be proved well defined, i.e., we have to show that a definition is not affected by a change in the representative (remember that the α-equivalence is an equivalence relation).

For example, consider the substitution of names. We said that it is mandatory to prevent the capture of bound names; however, how can we formalise this idea? It means that the term should be explored, and if the substituted name appears to be bounded, the subterm must be changed by another that is α-equivalent to the original. This change is a complex operation for the machine and contravenes the criteria given by de Bruijn. As a consequence, using this type of representation results in complex arguments, and it is common to require many obscure results.

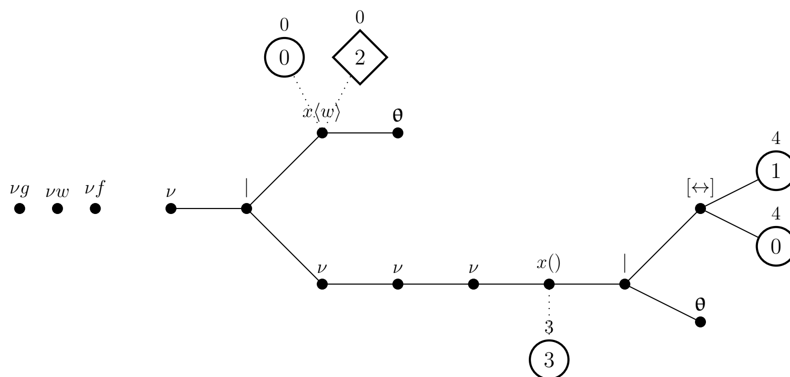
The second representation was proposed by de Bruijn in 1972 [de 72] and is known as de Bruijn indices. In this approximation, names are replaced by natural numbers, which reference the binder of the name. Here, we follow the convention used by Charguéraud and start counting the indices in 0. Consider the following term in the π-calculus:

$$\nu x.(x\langle w\rangle.\boldsymbol{\theta}|\nu z.\nu t.\nu s.x(y).([s \leftrightarrow y]|\boldsymbol{\theta})) \tag{2.1}$$



At first, it is necessary to fix a context from which the free names are taken, according to De Bruijn. De Bruijn used an ordered list as a context but also in the general case an infinite sequence of names (say $n_1, n_2, ...,$) works. The context for this example will be (f, w, g). Hence to obtain the de Bruijn representation, we apply the following steps:

1. At the left of the tree, draw binders for each name in the context in order.
2. For each name, compute the following:
   (a) The number of crossed binders in the way to the binder of the name (level).
   (b) The number of crossed binders in the way to the root of the tree (depth).
3. Erase the bounded names from all the binders ($x(y)$ and νy) in the tree.
4. Reconstruct the term with the aid of the tree, using the level number instead of the original name.



From this tree we get the translated term using de Bruijn notation:

$$\nu.(0\langle 2\rangle.\boldsymbol{\theta}|\nu.\nu.\nu.3(_).([1 \leftrightarrow 0]|\boldsymbol{\theta}))$$

From the latter expression and de Bruijn notation, some observations follow:

- each name has two numbers associated: the first is the level, which refers to the number of binders that the name must traverse to encounter its binder. The second is the depth, which is the number of binders that the name encounters on its way to the root of the tree. Thus a name is free in the term, if and only if its level is greater than its depth.
- de Bruijn puts the binder of free names to the left of the term tree. It should be noted that these binders are sensitive to the order of the context.
- In de Bruijn notation, the binder $x(y)$ is written $3(_)$. Due to a possible collision with terms like '$x()$.' which has another meaning in πULL.

De Bruijn notation solves a drawback from the naive strings notation, α-equivalence is no longer required, as a consequence of the first observation. Since the level is greater than the depth, any substitution of free names will never collide with a bound name, and the substitutions become safe. Unfortunately, there is a price to pay for this, requiring two new functions.

For example, Term 2.1 reduces to $\nu x.(\boldsymbol{\theta}|\nu z.\nu t.\nu s.([s \leftrightarrow w]|\boldsymbol{\theta}))$ which in de Bruijn notation (with context (f, w, g)) is $\nu.(\boldsymbol{\theta}|\nu.\nu.\nu.([0 \leftrightarrow 5]|\boldsymbol{\theta}))$. Note that the level of the name w must change, due to the new binders that are needed to traverse in order to reach the binder of w. Hence, the substitution when using de Bruijn indices needs a new definition. Suppose that the free names are indexed; in other words, fix a context $(x_1, x_2, ...)$; if n, m are the indices from the changing names, then:

$$\begin{aligned}
i\{m/n\} &= \begin{cases} m & n = i \\ i & \text{otherwise} \end{cases} \\
\boldsymbol{\theta}\{m/n\} &= \boldsymbol{\theta} \\
(P|Q)\{m/n\} &= (P\{m/n\})|(Q\{m/n\}) \\
(!P)\{m/n\} &= !(P\{m/n\}) \\
(i\langle j\rangle.P)\{m/n\} &= [i\{m/n\}]\langle j\{m/n\}\rangle.[P\{m/n\}] \\
(i(_).P)\{m/n\} &= [i\{m/n\}](_).[P\{m+1/n+1\}] \\
(\nu.P)\{m/n\} &= \nu.(P\{m+1/n+1\})
\end{aligned}$$

It is necessary to introduce the shift functions for de Bruijn indices. Here, the shift function is fused with the substitution function, in other words the above function is the substitution function and, at the same time, the shift function. The reason is that the substitution in π-calculus is different from the substitution in λ-calculus. In the last, it is possible to inject an expression inside another using β-rule ($(\lambda x.A)B \rightarrow_\beta A[x := B]$). In π-calculus, as a result of the formal definition of substitution, this behaviour is not needed. In π-calculus, names substitute other names and not terms.

De Bruijn indices suffer from two drawbacks [BU07, Kam01]: first is the complexity of implementing the shift functions; poor interpretations of these functions would lead to bad results or wrong reasoning. The second is related to the free names: How to choose the contexts? In general, this question has no answer and could lead to intense argumentation.

2.4 Locally Nameless Representation

Charguéraud [Cha12] proposed an alternative notation that is in the middle of the two previously discussed, the Locally Nameless Representation (LN); in this approach, the variables are tagged depending on their class (free or bound). Hence, the grammar is modified to include these tags.

Definition 2.4. *The following grammar generates the LN terms for the πULL processes:*

$$N = FName(x) \mid BName(i)$$
$$P, Q = \boldsymbol{\theta} \mid [N \leftrightarrow N] \mid P|Q \mid \nu.P \mid N\langle N\rangle.P \mid N(_).P \mid N().P \mid !N(_).P \mid N\langle\rangle.\boldsymbol{\theta}$$

where x represents a string, and i is a natural number. From now on, we write $x, y, z, ..$ for a free name, or $i, j, k, ...$ for a bounded a name.

For instance, the LN representation for Process 2.1 is:

$$\nu.(0\langle w\rangle.\boldsymbol{\theta}|\nu.\nu.\nu.3(_).([0 \leftrightarrow 1]|\boldsymbol{\theta}))$$

As we said, this representation is a reasonable compromise. With LN representation, there is no need for α-equivalence, and in particular, there is no risk of variable capture when doing a substitution. Furthermore, free names substitution is as usual, and there is no need to work with a particular context.

As usual, there is a price to pay for using this notation, and it is the definition of the open ($\{k \rightarrow x\}P$), and close ($\{k \leftarrow x\}P$) functions. Consider the following example: the term $P = x\langle y\rangle.\boldsymbol{\theta}$ has x as a free name, and we

need to bound it. If we just put a binder ν in front of P, x remains free in the result. Thus, we need a function that closes or changes a name for a pointer (reference).

On the other hand, suppose that there is a term of the form $\nu.P$, if we want to remove the binder, then we need an open function. Charguéraud [Cha12] comments about these functions:

> [Cha12, p. 5] Variable open turns some bound variables into free variables. It is used to investigate the body of an abstraction. Variable close turns some free variables into bound variables it is used to build an abstraction given a representation of its body.

Definition 2.5. *The open function takes a natural number (k), a string (x), a process (P) and replace each occurrence of a bound name with index k for the name x is defined as follows:*

$$
\begin{array}{lcl}
\{k \to x\}(i) & = & \begin{cases} x & i = k \\ i & otherwise \end{cases} \\
\{k \to x\}(y) & = & y \\
\{k \to x\}(\mathbf{0}) & = & \mathbf{0} \\
\{k \to x\}([N \leftrightarrow M]) & = & [\{k \to x\}N \leftrightarrow \{k \to x\}M] \\
\{k \to x\}(P|Q) & = & (\{k \to x\}P)|(\{k \to x\}Q) \\
\{k \to x\}(\nu.P) & = & \nu.(\{k+1 \to x\}P) \\
\{k \to x\}(N\langle M\rangle.P) & = & (\{k \to x\}N)\langle\{k \to x\}M\rangle.(\{k \to x\}P) \\
\{k \to x\}(N(_).P) & = & (\{k \to x\}N)(_).(\{k+1 \to x\}P) \\
\{k \to x\}(N().P) & = & (\{k \to x\}N)().(\{k \to x\}P) \\
\{k \to x\}(!N(_).P) & = & !(\{k \to x\}N)(_).(\{k+1 \to x\}P) \\
\{k \to x\}(N\langle\rangle.\mathbf{0}) & = & (\{k \to x\}N)\langle\rangle.\mathbf{0}
\end{array}
$$

Similarly, there is a close function. The close function takes a free name x and replaces its occurrences with index references.

$$\begin{array}{lcl}
\{k \leftarrow x\}(i) & = & i \\
\{k \leftarrow x\}(y) & = & \begin{cases} k & y = x \\ y & \textit{otherwise} \end{cases} \\
\{k \leftarrow x\}(\mathbf{0}) & = & \mathbf{0} \\
\{k \leftarrow x\}([N \leftrightarrow M]) & = & [\{k \leftarrow x\}N \leftrightarrow \{k \leftarrow x\}M] \\
\{k \leftarrow x\}(P|Q) & = & (\{k \leftarrow x\}P)|(\{k \leftarrow x\}Q) \\
\{k \leftarrow x\}(\nu.P) & = & \nu.(\{k+1 \leftarrow x\}P) \\
\{k \leftarrow x\}(N\langle M\rangle.P) & = & (\{k \leftarrow x\}N)\langle\{k \leftarrow x\}M\rangle.(\{k \leftarrow x\}P) \\
\{k \leftarrow x\}(N(_).P) & = & (\{k \leftarrow x\}N)(_).(\{k+1 \leftarrow x\}P) \\
\{k \leftarrow x\}(N().P) & = & (\{k \leftarrow x\}N)().(\{k \leftarrow x\}P) \\
\{k \leftarrow x\}(!N(_).P) & = & !(\{k \leftarrow x\}N)(_).(\{k+1 \leftarrow x\}P) \\
\{k \leftarrow x\}(N\langle\rangle.\mathbf{0}) & = & (\{k \leftarrow x\}N)\langle\rangle.\mathbf{0}
\end{array}$$

When $k = 0$ it is convenient to use the following notation.

$$\{0 \rightarrow x\}P := P^x \qquad\qquad \{0 \leftarrow x\}P := {}^xP$$

One may argue that these functions are troublesome, and it is better to use de Bruijn notation. Nevertheless, their implementation is easy, and any error is simple to find and fix; additionally, the argumentation with these functions is usually easier than with the shift functions.

Next, we present some technical results for the LN representation.

Lemma 2.14. *For all names N, natural number k, and free name x if $x \neq N$, then $\{k \leftarrow x\}(\{k \rightarrow x\}N) = N$.*

Proof. Using structural induction over N, there are two cases to consider:

- $N = y$. Using the hypothesis that $x \neq N$, hence $\{k \leftarrow x\}(\{k \rightarrow x\}y) = \{k \leftarrow x\}y = y$.
- $N = i$. If $i = k$ then $\{i \leftarrow x\}(\{i \rightarrow x\}i) = \{i \leftarrow x\}x = i$ as required. In the case that $i \neq k$, the name is not changed by the open function, thereby neither the close. □

Proposition 2.1. *Given a process P, a natural number k, and a fresh name (i.e., a name not free in the process) x for P, then $\{k \leftarrow x\}(\{k \rightarrow x\}P) = P$.*

Proof. This proof uses structural induction over the process P.

- $P = \mathbf{0}$. It follows using the definition of open/close functions.
- $P = [N \leftrightarrow M]$. As x is fresh in P, we can apply the previous lemma.
- $P = Q|S$. The structural induction hypothesis state that $\{k \leftarrow x\}(\{k \rightarrow x\}Q) = Q$ for processes Q, S, and natural number k. The result follows from the definition of the open/close functions.
- $P = N\langle M\rangle.Q$. The definitions of the open/close functions tell that is enough to verify the result for N, M, and Q to obtain the result for P. But, this follows from the previous lemma and the induction hypothesis over Q. The same idea applies for $P = N().P$ and $P = N\langle\rangle.\mathbf{0}$.
- $P = \nu.Q$. The application of the open/close functions yields $\{k \leftarrow x\}(\{k \rightarrow x\}(\nu.Q)) = \nu.(\{k+1 \leftarrow x\}(\{k+1 \rightarrow x\}Q))$, then the result follows using the induction hypothesis applied to Q.
- $P = N(_).Q$ or $P = !N(_).Q$. It follows combining the two previous arguments.

Then the result follows from structural induction over the process P. □

Corollary 2.1. *Given a process P and a fresh name x for P, then ${}^{x}(P^{x}) = P$.*

Proof. Using that ${}^{x}(P^{x}) = \{0 \leftarrow x\}(\{0 \rightarrow x\}P))$, it follows from the previous proposition. □

In the previous results, we needed a fresh names for a given process, but we can get a new name easily. If s is the concatenation of all free names in the process, then $x = ss$ is a fresh name.

The disadvantage of the LN representation is that the modified grammar introduces some terms that are nonsense. For example, $\nu.[0 \leftrightarrow 3]$, here 3 points to a non-existent binder. Hence we need a way to ensure that the terms are well-formed. Charguéraud proposed the *local closure* predicate (lc) to characterize the terms constructed correctly [Cha12].

Definition 2.6. *The local closure predicate characterizes the well-formed terms, and the following rules define it:*

$$\frac{}{lc(\mathbf{0})} \qquad \frac{}{lc(x)} \qquad \frac{}{lc([x \leftrightarrow y])} \qquad \frac{}{lc(x\langle\rangle.\mathbf{0})}$$

$$\frac{lc(P) \qquad lc(Q)}{lc(P|Q)} \qquad \frac{lc(P)}{lc(x\langle y\rangle.P)} \qquad \frac{lc(P)}{lc(x().P)}$$

$$\frac{\forall y,\ lc(P^y)}{lc(x(_).P)} \qquad \frac{\forall y,\ lc(P^y)}{lc(\nu.P)} \qquad \frac{\forall y,\ lc(P^y)}{lc(!x(_).P)}$$

Here x, y represent free names.

Using the lc predicate, we can verify that the term $\nu.[0 \leftrightarrow 3]$ is not well-formed. If it were well-formed, then for each free name x it should be the case that $\{x \to 0\}[0 \leftrightarrow 3] = [x \leftrightarrow 3]$ is well-formed, but this is false since 3 is never locally closed. In conclusion, $\nu.[0 \leftrightarrow 3]$ is not a well-formed term. This type of bound name that points to nowhere will be called an orphan.

Definition 2.7. *A process term P in LNR such that for any free name x satisfies that $lc(P^x)$ is called a body.*

For body terms, here is proposed an alternative view to Charguéraud [Cha12] original idea; he defines a body as a term process P for which it exists a finite list of names L, such that for every name not in L when we open the term with that name the result is locally closed. Yet, here we state that a body must obey the rule for all free names. In practice, it is better to open the process with a name that is not free in it, but there is no reason for a body to not be locally closed when it is open with all of its free names.

The locally closed predicate enables induction over its defining rules, which is useful when we want to propagate a property over the well-formed terms rather than over all the LN terms. This is another example of structural induction and it applies to different results.

Suppose that P is a locally closed term. If it does not begin with a binder, then it must be the case that we do not find any orphan when we read P. But, what happens when P starts with a binder? For example, $P = \nu.Q$, the definition says that if we read Q and we do not cross any binder, then all bound names must be zero. Suppose that we cross a binder reading Q, then all bound names must be at most one; and so on.

Following this idea, Charguéraud gives another characterisation of being locally closed called the *locally closed at* predicate [Cha12].

Definition 2.8. *The locally closed at predicate is defined inductively as:*

$$
\begin{aligned}
lca(i,k) &= i < k \\
lca(x,k) &= true \\
lca(\mathbf{0},k) &= true \\
lca([N \leftrightarrow M],k) &= lca(N,k) \quad \textbf{\textit{and}} \quad lca(M,k) \\
lca(P|Q,k) &= lca(P,k) \quad \textbf{\textit{and}} \quad lca(Q,k) \\
lca(N\langle\rangle.\mathbf{0},k) &= lca(N,k) \\
lca(N\langle M\rangle.P,k) &= lca(N,k) \quad \textbf{\textit{and}} \quad lca(M,k) \quad \textbf{\textit{and}} \quad lca(P,k) \\
lca(N().P,k) &= lca(N,k) \quad \textbf{\textit{and}} \quad lca(P,k) \\
lca(\nu.P,k) &= lca(P,k+1) \\
lca(N(_).P,k) &= lca(N,k) \quad \textbf{\textit{and}} \quad lca(P,k+1) \\
lca(!N(_).P,k) &= lca(N,k) \quad \textbf{\textit{and}} \quad lca(P,k+1)
\end{aligned}
$$

Now we present some properties of the lca predicate.

Lemma 2.15. *For any name N and natural number k, if $lca(N,k+1)$ then $\{k \to x_k\}...\{0 \to x_0\}N$ is a free name.*

Proof. Since the name is locally closed at $k+1$ then there are two possibilities: the first is that N is a free name and therefore $\{k \to x_k\}...\{0 \to x_0\}N = N$ which is a free name; in the second, N is a bound name $0 \leq i \leq k$, then the i-th operation replaced it by x_i and $\{k \to x_k\}...\{0 \to x_0\}N = x_i$. In both cases, the result is a free name. □

Proposition 2.2. *Given a process P, a natural number k and free names $x_0, ..., x_k$, if $lca(P,k+1)$ then $lc(\{k \to x_k\}...\{0 \to x_0\}P)$.*

Proof. The proof uses induction on P:

- $P = \mathbf{0}$. For any natural number p and free name x, it is true that $\{p \to x\}\mathbf{0} = \mathbf{0}$, then $\{k \to x_k\}...\{0 \to x_0\}\mathbf{0} = \mathbf{0}$ and it is locally closed.

- $P = [N \leftrightarrow M]$ or $P = N\langle\rangle.\mathbf{0}$. Lemma 2.15 asserts that $\{k \to x_k\}...\{0 \to x_0\}N$ ($\{k \to x_k\}...\{0 \to x_0\}M$) is a free name, and then P is locally closed.

- $P = Q \mid S$. From hypothesis, it follows that $lca(Q,k+1)$ and $lca(S,k+1)$, then using the induction hypothesis over Q and S, we obtain that $lc(Q)$, $lc(S)$, and these both imply that $lc(P)$.

- $P = N\langle M\rangle.Q$ or $P = N().Q$. Using the previous lemma we get that $\{k \to x_k\}...\{0 \to x_0\}N$ is a free name (M resp.), the induction hypothesis guarantees that $lc(\{k \to x_k\}...\{0 \to x_0\}P)$ and the result follows.

- $P = \nu.Q$. In this case the induction hypothesis is: for all $l \geq 0$ and free names $x_l, ..., x_0$, if $lca(Q, l + 1)$ then $lc(\{l \to x_l\}...\{0 \to x_0\}Q)$.

  On the other hand, as $lca(\nu.Q, k + 1) = lca(Q, k + 2)$ and $\{k \to x_k\}...\{0 \to x_0\}\nu.Q = \nu.(\{k + 1 \to x_k\}....\{1 \to x_0\}Q)$; then it is sufficient to show that for any free name y, $lc(\{0 \to y\}\{k+1 \to x_k\}...\{1 \to x_0\}Q)$ but using Lemma A.1 we can derive that $lc(\{k+1 \to x_k\}....\{1 \to x_0\}\{0 \to y\}Q)$ and the induction hypothesis with $l = k + 1$ and free names $y, x_0, ..., x_k$ yields what we need.

- $P =!N(_).Q$ or $P = N(_).Q$. We use Lemma 2.15 and the same idea of the previous case. □

Lemma 2.16. *For a name N and a natural number K. If $lca(\{k \to x\}N, k)$ for any free name x, then $lca(N, k + 1)$.*

Proof. There are two cases. If $N = y$ for a free name y, then it is locally closed at $k + 1$. If $N = i$ for a bound variable it is not the case that $k < i$ because $\{k \to x\}N = i$ and it needs to be lca at k. Thus, it is the case that $i \leq k$, which implies $lca(N, k + 1)$. □

Proposition 2.3. *For any process P and natural number k, if $lca(\{k \to x\}P, k)$ for any free name x, then $lca(P, k + 1)$.*

Proof. By induction over the structure of P:

- $P = \mathbf{0}$. It is true that $lca(P, k)$ for any natural number k.

- $P = [N \leftrightarrow M]$ or $P = N\langle\rangle.\mathbf{0}$. Use Lemma 2.16 to obtain that $lca(N, k + 1)$ and $lca(M, k + 1)$.

- $P = Q \mid S$. Using the induction hypothesis, obtain $lca(Q, k + 1)$ and $lca(S, k + 1)$.

- $P = N().Q$ or $P = N\langle N\rangle.Q$. From the hypothesis it is true that $lca(N, k)$ and $lca(Q, k)$, thus from Lemma 2.16 and induction hypothesis is true $lca(N, k + 1)$ and $lc(Q, k + 1)$.

- $P = \nu.Q$. The hypothesis states that for any free name x it is the case that $lca(\{k \to x\}\nu.Q, k)$ which is the same as $lca(\nu.\{k+1 \to x\}Q, k) = lca(\{k + 1 \to x\}Q, k + 1)$.

On the other hand, the induction hypothesis is: for any natural number k', if it is true that $lca(\{k' \to x\}Q, k')$ for any free name x then $lca(Q, k'+1)$.

Use the hypothesis and the induction hypothesis to obtain the result.

- $N(_).P$ or $!N(_).P$. Use the previous arguments. □

So far, we introduced the necessary lemmas to prove the equivalence between lc and lca. Now, we present an important equivalence which is only stated by Charguéraud [Cha12].

Theorem 2.3. *For any LNR term P, $lc(P)$ if and only if $lca(P, 0)$.*

Proof. First, there is the proof that if the term is locally closed, then it is locally closed at 0. For this, we use induction over the locally closed predicates.

- $lc(\mathbf{0})$. $\mathbf{0}$ is locally closed at any k.
- $lc([x \leftrightarrow y])$ or $lc(x\langle\rangle.\mathbf{0})$ .As the free names are always locally closed, the result follows.
- $lc(Q|S)$. From the definition of locally closed, $lc(Q)$, and $lc(S)$. Then applying the induction hypothesis, it is true that $lca(Q, 0)$, $lca(S, 0)$, and the result follows.
- $lc(x\langle y\rangle.P)$ or $lc(x().P)$. The free names are locally closed at 0, and the induction hypothesis states that $lca(P, 0)$. Thus in both scenarios, the claim follows.
- $lc(\nu.Q)$. To prove that $lca(\nu.Q, 0)$ is sufficient to show $lca(Q, 1)$. From the fact that for any free name x, $lc(Q^x)$, then $lca(Q^x, 0)$, and applying Proposition 2.3 it follows that $lca(Q, 1)$.
- $lc(x(_).P)$ or $lc(!x(_).P)$. Combine the previous cases.

Before proving the reverse direction, observe that for any name N if $lca(N, 0)$, it must be the case that N is a free; see Lemma A.3. Now the proof proceeds using induction over P:

- $P = \mathbf{0}$. The zero process is locally closed.
- $P = [N \leftrightarrow M]$ or $P = N\langle\rangle.\mathbf{0}$. By the initial observation, N and M are free names, therefore P is locally closed.

- $P = Q \mid S$. Using the induction hypothesis and the fact that $lca(Q, 0)$ ($lca(S, 0)$ resp.) then $lc(Q)$ ($lc(S)$ resp.), as a consequence $lc(P)$.
- $P = N().Q$ or $P = N\langle M\rangle.Q$. From the hypothesis, $lca(N, 0)$ and $lca(Q, 0)$, thus from the initial observation N is a free name, and using the induction hypothesis it is true that $lc(Q)$ and $lc(P)$. The same idea works for the other case.
- $P = \nu.Q$. From the hypothesis, $lca(Q, 1)$. Lemma 2.2 finishes the proof.
- $N(_).P$ or $!N(_).P$. The proof combines the previous argumentation and the initial observation. □

With the notion of a locally closed process, some definitions and results of π-calculus processes need modifications to capture the new ideas. Consider the reduction rules; how they should be when using LNR? The Open/Close operations play a fundamental role in the representation. Additionally, for most of the rules, the components require new conditions.

Now, definitions of congruence and reduction of terms can be defined using LN representation.

Definition 2.9. *Given two processes, P and Q, they are structural congruent if it is possible to transform one into the other using the following equations (in either direction):*

$$P|Q \equiv Q|P \qquad\qquad P|\nu.Q \equiv \nu.(P|Q),\ \textit{if } lc(P)$$

Definition 2.10. *Reduction of terms ($\longrightarrow$) is defined by:*

$$\frac{x \in fn(P),\ lc(P)}{\nu.\{0 \leftarrow x\}(P \mid [x \leftrightarrow y]) \longrightarrow P\{y/x\}}\ \textit{(fuse)}$$

$$\frac{x \in fn(P),\ lc(P)}{\nu.\{0 \leftarrow x\}(P \mid [y \leftrightarrow x]) \longrightarrow P\{y/x\}}\ \textit{(fuse)}$$

$$\frac{x \in fn(P),\ lc(P)}{\nu.\{0 \leftarrow x\}([x \leftrightarrow y] \mid P) \longrightarrow P\{y/x\}}\ \textit{(fuse)}$$

$$\frac{x \in fn(P),\ lc(P)}{\nu.\{0 \leftarrow x\}([y \leftrightarrow x] \mid P) \longrightarrow P\{y/x\}}\ \textit{(fuse)}$$

$$\frac{x \notin fn(P),\ lc(P)}{\nu.\{0 \leftarrow x\}(x\langle\rangle.\mathbf{0} \mid x().P) \longrightarrow P}\ \textit{(1)}$$

$$\frac{x \notin fn(P),\ lc(P)}{\nu.\{0 \leftarrow x\}(x().P \mid x\langle\rangle.\mathbf{0}) \longrightarrow P}\ \textit{(1)}$$

$$\frac{u, y \notin fn(P),\ Body(P),\ lc(Q)}{\nu.\{0 \leftarrow u\}(!u(_).P \mid \nu.\{0 \leftarrow y\}(u\langle y\rangle.Q)) \longrightarrow \nu.\{0 \leftarrow u\}(!u(_).P \mid \nu.\{0 \leftarrow y\}(Q \mid \{0 \rightarrow y\}P))}\ \textit{(Copy)}$$

$$\frac{u, y \notin fn(P),\ Body(P),\ lc(Q)}{\nu.\{0 \leftarrow u\}(!u(_).P \mid \nu.\{0 \leftarrow y\}(u\langle y\rangle.Q)) \longrightarrow \nu.\{0 \leftarrow u\}(!u(_).P \mid \nu.\{0 \leftarrow y\}(\{0 \rightarrow y\}P \mid Q))}\ \textit{(Copy)}$$

$$\frac{y \notin fn(P),\ y \notin fn(R),\ Body(P),\ lc(Q),\ lc(R)}{\nu.\{0 \leftarrow x\}(x(_).P \mid \nu.\{0 \leftarrow y\}(x\langle y\rangle.(Q \mid R))) \longrightarrow \nu.\{0 \leftarrow x\}(\nu.\{0 \leftarrow y\}(Q \mid \{0 \rightarrow y\}P) \mid R)}\ \textit{(M. Disjuction)}$$

$$\frac{y \notin fn(P),\ y \notin fn(R),\ Body(P),\ lc(Q),\ lc(R)}{\nu.\{0 \leftarrow x\}(\nu.\{0 \leftarrow y\}(x\langle y\rangle.(Q \mid R)) \mid x(_).P) \longrightarrow \nu.\{0 \leftarrow x\}(R \mid \nu.\{0 \leftarrow y\}(Q \mid \{0 \rightarrow y\}P))}\ \textit{(M. Conjunction)}$$

In previous definitions, the processes required new conditions, conditions that are crucial when one tries to formalize the system using Coq. Finally, relative to the locally closed relation, good behavior is expected for the reduction and congruence relations. In other words, we require that definitions preserve the locally closed property.

Theorem 2.4. *Given a locally closed process P, if $P \longrightarrow Q$ or $P \equiv Q$, then Q is locally closed.*

Proof. By induction on the reduction/congruence rule, using the following properties, which are presented on the appendix, and Theorem 2.3.

- If P is locally closed, then $\{k \rightarrow x\}P = P$ for every natural k and free name x.
- If P is locally closed, then $P\{y/x\}$ is locally closed for every pair of free names x, y.
- If P is a body, then $\{0 \rightarrow x\}P$ is locally closed for every free name x. □

In the first part of this chapter, we presented our version of πULL calculus. Furthermore, we show how to construct the system from scratch using

principal cuts. The construction given here helped us to implement to give a proof of the Subject Reduction Theorem without any hidden detail.

In the second part of this chapter, we show how to represent the πULL system using LN representation. We extended the ideas given by Charguéraud for our calculus, and we proved useful results of the representation. In particular, we gave a proof for the *lc* and *lca* equivalence.

3

Design and Coq implementation

Informal ideas are usual in mathematical works and within proofs. For example, phrases like: *analogous to the previous case, induction over P, etc.* Furthermore, it is common to encounter ideas that are easy to understand for humans but not for machines and viceversa, consider Lemma 2.15. Using proof assistants, like Coq, it is possible to us to exhibit these kinds of situations.

Studying these behaviours, challenges, questions, and problems is important for all the community leading to new conceptions about how theories are developed and making them more understandable.

In this final chapter, we discuss this phenomena in the context of the present work. We show some examples of challenging results, and tell how we solved the difficulties in the implementation. We intent to show that in some cases the solutions are nontrivial, and require thinking outside the box.

From now, we use the notation from Table A.2.5 in the appendix.

3.1 Interactions between operations

The locally nameless representation introduces new operations, the most important of them are the open and close operations. On the other hand, there are operations from the theory of π-calculus such as the substitution. We need to know how these operations interact. in order to complete the verification.

We previously established that it holds for locally closed terms:

- It is always true that, $\{0 \to x\}\{0 \leftarrow x\}P = P$.
- If $x \notin fn(P)$, then $\{0 \leftarrow x\}\{0 \to x\}P = P$.

Note that we used some side conditions to have good behaviour when composing two operations. Nevertheless, some interactions will require more

than a restriction in the process. For example, the substitution-close interactions or the injectivity of the close operator.

Consider the following process $\nu x.P$, so if one writes $(\nu x.P)\{u/v\}$ there are some assumptions that have to be made: the first is that there will be no variable capture, in other words, $u \neq x$; and the second is that the substitution does not change the bound variable, $v \neq x$.

Now, consider the term $\nu.\{0 \leftarrow x\}P$ in LNR, and we want to perform the substitution $(\nu.\{0 \leftarrow x\}P)\{u/v\}$. In this case, we expect the same properties of this substitution as in the π-calculus.

The implementation and verification of these properties in Coq is not trivial and does not follow from the definitions, so it requires management at the implementation level. The following inductive definition helps to formalize the idea.

```
Inductive IsClosing : Process → nat → Prop :=                              1
  | IsClosing_Base : forall ( P : Process)(x: nat),                         2
    (forall (u v : nat)(Q : Process), Q = ({u \ v} Close x P) → u ≠x ∧ x ≠v  3
       ) → (IsClosing P x).
#[global]                                                                   4
Hint Constructors IsClosing : Piull.                                        5
```

This definition aims at concluding that within a close-substitution composition it is possible to assume that the names are different. Despite this, note that this definition is, in some sense an axiom, as it is possible to establish the equality in the antecedent part and get the consequent. But, it differs from an axiom in the sense that the user (mathematician) writing the proof is responsible for using or not using the definition, so it is not possible to use it accidentally.

The following problem arises when we use equality of terms under α-equivalence. Consider that $\nu x.P =_\alpha \nu y.Q$, then it is possible to make finite changes in bounded names of $\nu y.Q$ and find P' such that $\nu x.P = \nu x.P'$. So it is always possible to extract the equality $P = P'$.

In the case of the LNR operations, a similar situation arises. Consider the following equation $\{0 \rightarrow x\}P = \{0 \rightarrow y\}Q$; there are two possibilities:

- If $x = y$, then it should be the case that $P = Q$.
- If $x \neq y$, then it cannot be said anything. For example, consider $\{0 \rightarrow x\}[x \leftrightarrow z] = \{0 \rightarrow y\}[y \leftrightarrow z]$.

Hence, it is desirable in some situations to have the ability to use the α-equivalence argument. The following definition enables it.

```
Inductive IsClosingInj : Process → nat → Prop :=                           1
```

```
  | IsClosingInj_Base : forall ( P : Process)(x: nat),
    (forall (u : nat)(Q : Process), Q = (Close x P) → u = x ) → (
        IsClosingInj P x).
#[global]
Hint Constructors IsClosingInj : Piull.
```

Note that as in the previous case using the definition is the responsibility of the user who writes the proof. So there is no possibility to use it accidentally.

3.2 Locally Closed and Locally Closed At

The LN representation has two predicates lc and lca. The predicates verify when a term is correct from construction within the representation. Nevertheless, the idea that supports each one is different; the natural numbers and bounds guide the lca predicate, and the idea of being well-open is implemented by the lc predicate.

For most lemmas and theorems in LNR, the proof has two parts: one lemma for the names and one for the processes. In most cases, the results can be derived using structural induction over the process, and sometimes over the well-formed property. To illustrate this, consider the following:

Property 3.1. *Given a process P, natural number k, and free name y if $lc(P)$, then $lc(\{k \to y\}P)$.*

The property states that opening a bound name in a well-formed term does not modify the well-formed property. To prove this result, we use the following property on names:

Property 3.2. *For all free names x, y, and natural number k. If $lc(x)$, then $\{k \to y\}x = x$.*

The proof of Property 3.1 is by induction on the lc property on P and a sketch of the argument is:

- If $P = \mathbf{0}$, the result follows by definition.
- If $lc(P)$ is $[x \leftrightarrow y]$ or $x\langle\rangle.\mathbf{0}$ or $Q \mid S$; the result follows follows by induction and (or) applying the property for the names.
- If $\nu.P$, the induction hypothesis solves the case.
- For the other cases, we combine the previous ones.

This same proof structure applies to many of the results. Now, consider:

Property 3.3. *Given a process P and free names y, x; if $lc(P)$, then $lc(\{y/x\}P)$.*

If we try to use induction over the lc property in the $\nu.P$ case, to apply the induction hypothesis we need P to be locally closed, but this is always false (P is a body). As a consequence, the proof is stuck.

On the other hand, if we first apply Theorem 2.3 to obtain that P is locally closed at zero, then the proof can be done using induction over P.

As illustrated, the equivalence between the lc and lca is relevant to the representation; it proves many results in the present work. So consider the following informal reasoning:

Theorem 3.1. *For any LNR term P, $lc(P)$ if and only if $lca(P, 0)$.*

Proof. In one direction, as the term is lc then it is the case that every bound name is associated with a binder, so there is no possibility to have any bound name greater than zero, in other words, it is lca at zero. The other direction follows from the idea that if the process is lca at level k then one could open the process with k names, and the result is a process with all the bound names pointing to some binder, i.e. a lc term; as the process is lca at level zero, the result follows. □

To formalise the proof, we need to implement the idea of multiple opens, i.e. $\{i_k \to x_k\}...\{0 \to x_0\}P$.

```
Fixpoint MOpen_Rec (k : nat)(L : list nat)( T : Process ) : Process :=
match L , k with
  | nil , _  ⇒ T
  | x ::  L0, 0 ⇒ { 0 → x } (MOpen_Rec 0 L0 T)
  | x ::  L0, S t ⇒ { t → x } (MOpen_Rec t L0 T)
end.
#[global]
Hint Resolve MOpen_Rec : Piull.
```

The main advantage of this definition is that many results involving induction on the structure of the process are translated into induction on the length of the list; and induction on lists is easier and does not require additional results. The following theorem shows the convenience of the multiple opens to prove the equivalence.

```
Theorem Lca_Lc_Process_MOpen :
forall (P : Process)(k : nat)(L : list nat),
(length L) = k →
lca k P → lc (MOpen_Rec k L P).
```

With this property, the proof of the equivalence is similar to the one presented in Theorem 2.3.

3.3 Multiple Open

Our motivation for multiple opens is to make easier the proof of relevant results. For example, consider the following property.

Property 3.4. *Given a process $x \cdot P$ that is lca at level k, then $\{i_k \rightarrow x_k\}...\{0 \rightarrow x_0\}(x \cdot P) = y \cdot \{i_k + 1 \rightarrow x_k\}...\{1 \rightarrow x_0\}P$ for some free name y.*

This property has in Coq the following signature:

```
Lemma MOpen_Chan_zero_NoMOpenName :
forall (k : nat)(L : list nat)(x : Name),
(length L) = k → lca k ( x · P) →
exists (y : nat), MOpen_Rec k L ( x · P) = (FName y) · (MOpen_Rec+1 k L P).
```

To prove this, a difficult induction is required, and the recursive structure of the process is wasted. Furthermore, the predicate MOpen_Rec+1 implementing the shifted openings and needed when a binder is crossed, was not implemented due to complications in the design. So, we took another path, this path is easier and helps to structure a cleaner and more understandable proof. The idea is to take advantage of the recursive structure; we divide the implementation of the multiple openings into three parts:

- Keep the definition MOpen_Rec as the definition for $\{i_k \rightarrow x_k\}...\{0 \rightarrow x_0\}P$.
- Implement a definition for multiple openings in the case of names, x.
- Implement a definition of multiple openings M2Open_Rec, $\{i_k + 1 \rightarrow x_k\}...\{1 \rightarrow x_0\}P$, which takes care of the shifting.

With this new approach, the recursive structure is exploited, and the focus is on proving the following:

```
Lemma MOpen_Name_Result :
forall (k : nat)(L : list nat)(x : Name),
(length L) = k → lca_name k x →
exists (x0 : nat), ( MOpen_Name_Rec k L x) = FName x0.
```

Hence, the previous theorem can be stated as follows:

```
Lemma MOpen_Chan_input :
forall (k : nat)(L : list nat)(x : Name)(P : Process),
(length L) = k →
MOpen_Rec k L (x · P) = (MOpen_Name_Rec k L x) · (M2Open_Rec k L P).
```

This is easier to prove than the first version.

The new implementation helps to exploit the recursive structure to generate relations between the multiple openings. For example,

```
Lemma M2Open_MOpen :
forall (k x : nat)(L : list nat)(P : Process),
(length L) = k →
({0 → x} M2Open_Rec k L P) = MOpen_Rec (S k) (L + + (x :: nil)) P.
```

One open question we left for future work is: How to implement in a more general way the notion of multiple openings in Coq? Note that if one requires to work with openings shifting 4 places, it requires an implementation like:

```
Fixpoint M4Open_Rec (k : nat)(L : list nat)( T : Process ) : Process :=
match L , k with
  | nil , _  ⇒ T
  | x ::  L0, 0 ⇒ { 0 → x } (M4Open_Rec 0 L0 T)
  | x ::  L0, (S 0) ⇒ { (S 0) → x } (M4Open_Rec 0 L0 T)
  | x ::  L0, (S (S 0)) ⇒ { (S (S 0)) → x } (M4Open_Rec 0 L0 T)
  | x ::  L0, S t ⇒ { S t → x } (M4Open_Rec t L0 T)
end.
#[global]
Hint Resolve M4Open_Rec : Piull.
```

But dealing with the relations between the different shiftings would require many results. Another disadvantage of the current implementation is that its correct use is in the user's hands and is prone to errors. For example, calling the M4Open_Rec with just two elements in the list. So, it is necessary to search for one that works for an arbitrary number of shiftings.

3.4 Lists or Sets

Another key structure within the project is the notion of Context. The difficulty comes in choosing the correct structure to represent them. The two most known are List or Ensembles, each having their advantages or disadvantages; the idea here is to discuss the reason why the Ensembles structure was the chosen one.

The first drawback found working with lists is the position of the assignments that are being modified by the rules, for example, the CutR:

$$\frac{\Gamma; \Delta \vdash P :: \Lambda, x : A \qquad \Gamma; \Delta', x : A \vdash Q :: \Lambda'}{\Gamma; \Delta, \Delta' \vdash \nu x.(P|Q) :: \Lambda, \Lambda'} \text{(CutR)}$$

In this rule, the assignment $x : A$ is at the tail of both lists; look at the upper sequents of the rule. But, what about the other possibilities? Cannot the assignment be at the head of the lists? This discussion leads to considering different locations for the assignments in the different rules, and there are many possibilities to make a decision.

Suppose that one agrees to have all the different positions for the rules. Consider as a second example the transference property in the case of the Λ context. If the implementation of the context uses lists then it is necessary to have four transference properties, depending on the location and destination of the assignment:

O/ D	Head	Tail
Head	T.1	T.2
Tail	T.2	T.3

Other branches of the project include the exploration of other implementations, such as Snoc Lists, but this suffers from the same disadvantages of Lists. In the project's branches:

- master.
- snoc.
- set-reglas.

we explore different context structures.

3.5 Unspoken words

One key difference between the written work and the source code is the detail level that the second has. The first step in the implementation of the Coq verification is to settle all the notions and ideas that are never written but are implicitly used in some works. For example, consider a sequent $\Gamma; \Delta \vdash P :: \Lambda$. Many questions arise around them:

- Can an assignment be in $\Gamma \cap \Delta$, $\Gamma \cap \Lambda$, or $\Delta \cap \Lambda$?
- Does it make sense that $x : A \in \Delta$ and $x : B \in \Delta$ being $A \neq B$?
- If we write $x : A, \Delta$, can we consider that $x : A \notin \Delta$?

These questions are common during the process of verification and have impact in the implementation time. The answer to these questions are conditions over the contexts implementation. For example, within the typing rule it is necessary to have the following predicate:

```
Inductive Good_Contexts : Context → Context → Context → Process →
    Prop := is_well_collected :
forall (D F G : Context)(P : Process),
  ( forall (x : nat), ( (x ∈ FVars P) → ( exists (A : Proposition), (FName x
      :A) ∈ (D ∪ F ∪ G) ) )) ∧
  ( (Disjoint_Sets D F) ∧ (Disjoint_Sets D G) ∧ (Disjoint_Sets F G) ∧ (
      Injective D) ∧ (Injective F) ∧ (Injective G))
   → (Good_Contexts D F G P).
```

The predicate controls the behavior of the context, so there are not noisy cases in the verification. In the case of the **1**L typing rule:

```
onel : forall ( D F G : Context )( x : nat )( P : Process ),
    Collect D → Collect F → Collect G → lc P →
    Good_Contexts D F G P →
    Good_Contexts D ( (Bld x 1 ) ∪ F ) G (FName x ()· P ) →
    ( D ;;;  F ⊢ P :::  G ) →
    ( D ;;;  ( (Bld x 1 ) ∪ F ) ⊢ (FName x ()· P ) :::  G )
```

It is not the case that x is in D or G, and by the Good_Context predicate, it is neither the case that is in F. So, good behavior of the sequent is required before and after typing.

The second example comes from the process implementation. Previously, we defined the following LN representation rule:

$$\frac{u, y \notin fn(P),\ Body(P),\ lc(Q)}{\nu.\{0 \leftarrow u\}(!u(_).P \mid \nu.\{0 \leftarrow y\}(u\langle y\rangle.Q)) \longrightarrow \nu.\{0 \leftarrow u\}(!u(_).P \mid \nu.\{0 \leftarrow y\}(Q \mid \{0 \rightarrow y\}P))}\ \text{(fuse)}$$

Its Coq implementation requires to make more details explicit. For example $u \neq y$, the predicates IsClosing, and IsClosingInj for dealing with substitutions, and the explicit mention that u and y are free names. Note that this differs from the pure paper rule given in Section 2.1.

```
| Red_parallel_replicate_rg : forall (y u : nat) (P Q : Process),
    Body P → lc Q → ¬ u ∈ FVars P → ¬ y ∈ FVars P → u ≠y → IsClosing P
        u →
    IsClosing P y → IsClosingInj P u → IsClosingInj P y →
    ( ν Close u ( ((FName u) !.  P) |   ν  Close y  (  ((FName u) 〈 FName y〉 . Q))
        )
        ⟶ ν Close u ( ((FName u) !. P) |   ν  Close y  (  Q |  ({0 → y} P) )) )
```

This example illustrates the difference when the LN representation comes into scene. The LN representation requires more conditions, as it is the case of the IsClosing predicates. This kind of conditions are not part of the original formulation of the rule but necessary to complete the verification.

4

Final remarks

In this work we pursued two objectives:

- We sought after a presentation in detail of the Subject Reduction Theorem for πULL system.
- We sought after a formal verification of the Subject Reduction Theorem in Coq.

To complete the first goal, we start reviewing existing works on session types, since we wanted to understand all the notions around them. In particular, we focused on Heuvel and Perez [vdHP20] and Caires et. al. [CPT12] works. Heuvel's work helped us to design our proper version of the πULL type system, and Caires' work provided us with the tools and ideas to validate the system and its rules.

In Chapter 1, we presented all the previous notions and ideas required to understand the session types system. We made a detailed exposition, covering details that are omitted by other authors. We filled up found gaps that would be problematic for our second goal.

In Chapter 2, we completed the proof of the Subject Reduction Theorem. We developed a traditional paper and pencil proof including particular details and covering all the auxiliary lemmas that we found during the Coq implementation phase. Further, we intended to make the implementation straightforward, and clear as possible.

We found many challenges during the formal verification process in Coq. For example, in Section 2.2 we discussed the reasons and motivation that lead us to choose Charguéraud's [Cha12] locally nameless representation. Furthermore, the study of the LN representation faced us with another other challenges; the most important is the equivalence of the predicates lc and lca, for which we gave a comprehensive proof.

Finally, in the last chapter, we explore some relevant questions that arose during the development of the work. Most of these questions remain open, and we are sure that they questions required a non-trivial independent work.

The nature of these questions is different and intersect different disciplines. For us, the most important questions are:

- The advantages and drawbacks of each context representation. Can we define a better representation for them?
- How can we reconcile the calculus operations and the LN representation operations?

In conclusion, we approached our objectives with a holistic point of view, these helped us to understand the πULL system, and to solve our goals. We found many challenges, however, we faced them and solved them with original ideas. We expect this work would help the community to improve its understanding of session type systems, as well awake the curiosity for the formal verification ideas and developments.

A

The Appendix

A.1 πULL type system rules.

The typing rules for the system are:

$$\frac{}{\Gamma; x : A, y : A^{\perp} \vdash [x \leftrightarrow y] ::}\ (\text{IdL}) \qquad \frac{}{\Gamma; x : A \vdash [x \leftrightarrow y] :: y : A}\ (\text{IdR})$$

$$\frac{\Gamma; \Delta \vdash P :: \Lambda, x : A \qquad \Gamma; \Delta', x : A \vdash Q :: \Lambda'}{\Gamma; \Delta, \Delta' \vdash \nu x.(P|Q) :: \Lambda, \Lambda'}\ (\text{CutR})$$

$$\frac{\Gamma; \Delta, x : A \vdash P :: \Lambda \qquad \Gamma; \Delta', x : A^{\perp} \vdash Q :: \Lambda'}{\Gamma; \Delta, \Delta' \vdash \nu x.(P|Q) :: \Lambda, \Lambda'}\ (\text{CutL})$$

$$\frac{\Gamma; x : A^{\perp} \vdash P :: \cdot \qquad \Gamma, u : A; \Delta \vdash Q :: \Lambda}{\Gamma; \Delta \vdash \nu u.(!u(x).P|Q) :: \Lambda}\ (\text{CutR?})$$

$$\frac{\Gamma; \cdot \vdash P :: x : A \qquad \Gamma, u : A; \Delta \vdash Q :: \Lambda}{\Gamma; \Delta \vdash \nu u.(!u(x).P|Q) :: \Lambda}\ (\text{Cut!})$$

$$\frac{\Gamma, u : A; \Delta, x : A \vdash P :: \Lambda}{\Gamma, u : A; \Delta \vdash \nu x.u\langle x \rangle.P :: \Lambda}\ (\text{CopyL})$$

$$\frac{\Gamma, u : A; \Delta \vdash P :: \Lambda, x : A^{\perp}}{\Gamma, u : A; \Delta \vdash \nu x.u\langle x \rangle.P :: \Lambda}\ (\text{CopyR})$$

$$\frac{\Gamma; \Delta \vdash P :: \Lambda}{\Gamma; \Delta, x : 1 \vdash x().P :: \Lambda}\ (\text{1L}) \qquad \frac{}{\Gamma; \cdot \vdash x\langle\rangle.\mathbf{0} :: x : 1}\ (\text{1R})$$

$$\frac{}{\Gamma; x :\perp \vdash x\langle\rangle.\mathbf{0} :: \cdot}\ (\perp L) \qquad \frac{\Gamma; \Delta \vdash P :: \Lambda}{\Gamma; \Delta \vdash x().P :: \Lambda, x :\perp}\ (\perp P)$$

$$\frac{\Gamma; \Delta, y : A, x : B \vdash P :: \Lambda}{\Gamma; \Delta, x : A \otimes B \vdash x(y).P :: \Lambda}\ (\otimes L)$$

$$\frac{\Gamma; \Delta \vdash P :: \Lambda, y : A \qquad \Gamma; \Delta' \vdash Q :: \Lambda', x : B}{\Gamma; \Delta, \Delta' \vdash \nu y.x\langle y\rangle.(P|Q) :: \Lambda, \Lambda', x : A \otimes B}\ (\otimes R)$$

$$\frac{\Gamma; \Delta, y : A \vdash P :: \Lambda \qquad \Gamma; \Delta', x : B \vdash Q :: \Lambda'}{\Gamma; \Delta, \Delta', x : A \parr B \vdash \nu y.x\langle y\rangle.(P|Q) :: \Lambda, \Lambda'}\ (\parr L)$$

$$\frac{\Gamma; \Delta \vdash P :: \Lambda, x : B, y : A}{\Gamma; \Delta \vdash x(y).P :: \Lambda, x : A \parr B}\ (\parr R)$$

$$\frac{\Gamma, u : A; \Delta \vdash P :: \Lambda}{\Gamma; \Delta, x :!A \vdash P\{x/u\} :: \Lambda}\ (!\mathrm{L}) \qquad \frac{\Gamma; \cdot \vdash P :: y : A}{\Gamma; \cdot \vdash !x(y).P :: x :!A}\ (!\mathrm{R})$$

$$\frac{\Gamma; y : A \vdash P :: \cdot}{\Gamma; x :?A \vdash !x(y).P :: \cdot}\ (?\mathrm{L}) \qquad \frac{\Gamma, u : A; \Delta \vdash P :: \Lambda}{\Gamma; \Delta \vdash P\{x/u\} :: \Lambda, x :?A}\ (?\mathrm{R})$$

The following are the principal cut cases. The first is the identity rule.

$$\dfrac{\Gamma; \Delta \vdash P :: \Lambda, x : A \qquad \dfrac{}{\Gamma; x : A \vdash [x \leftrightarrow y] :: y : A}\,(\text{IdR})}{\Gamma; \Delta \vdash \nu x.(P \mid [x \leftrightarrow y]) :: \Lambda, y : A}$$

whose reduction rule is: $\nu x.(P \mid [x \leftrightarrow y]) \longrightarrow P\{y/x\}$.

$$\dfrac{\dfrac{}{\Gamma; x : A \vdash [x \leftrightarrow y] :: y : A}\,(\text{IdR}) \qquad \Gamma; \Delta', y : A \vdash Q :: \Lambda'}{\Gamma; \Delta', x : A \vdash \nu y.([x \leftrightarrow y] \mid Q) :: \Lambda'}$$

whose reduction rule is: $\nu y.([x \leftrightarrow y] \mid Q) \longrightarrow Q\{x/y\}$.

$$\dfrac{\dfrac{}{\Gamma; x : A, y : A^{\perp} \vdash [x \leftrightarrow y] :: \cdot}\,(\text{IdL}) \qquad \Gamma; \Delta', x : A^{\perp} \vdash Q :: \Lambda'}{\Gamma; y : A^{\perp}, \Delta' \vdash \nu x.([x \leftrightarrow y]|Q) :: \Lambda'}$$

whose reduction rule is: $\nu x.([x \leftrightarrow y]|Q) \longrightarrow Q\{y/x\}$.

$$\dfrac{\Gamma; \Delta, y : A \vdash P :: \Lambda \qquad \dfrac{}{\Gamma; x : A, y : A^{\perp} \vdash [x \leftrightarrow y] :: \cdot}\,(\text{IdL})}{\Gamma; x : A, \Delta \vdash \nu y.(P \mid [x \leftrightarrow y]) :: \Lambda}$$

whose reduction rule is: $\nu y.(P \mid [x \leftrightarrow y]) \longrightarrow P\{x/y\}$.

There are some cases for the identity rules that are not considered, for example composing the left identity and the left cut gives a rule in which a resource changes side. In other cases, we cannot make sense of it, for example, cuts involving identity and replicate or WhyNot cuts. Next, consider the Copy rules.

$$\dfrac{\Gamma; x : A^{\perp} \vdash P :: \cdot \qquad \dfrac{\Gamma, u : A; \Delta \vdash Q :: \Lambda, y : A^{\perp}}{\Gamma, u : A; \Delta \vdash \nu y.u\langle y\rangle.Q :: \Lambda}\,(\text{CopyR})}{\Gamma; \Delta \vdash \nu u.(!u(x).P \mid \nu y.u\langle y\rangle.Q) :: \Lambda}\,(\text{Cut?})$$

whose reduction rules is: $\nu u.(!u(x).P \mid \nu y.u\langle y\rangle.Q) \longrightarrow \nu u.(!u(x).P \mid \nu y.(Q \mid P\{y/x\}))$.

$$\dfrac{\Gamma; x : A^{\perp} \vdash P :: \cdot \qquad \dfrac{\Gamma, u : A; \Delta, y : A \vdash Q :: \Lambda}{\Gamma, u : A; \Delta \vdash \nu y.u\langle y\rangle.Q :: \Lambda}\,(\text{CopyR})}{\Gamma; \Delta \vdash \nu u.(!u(x).P \mid \nu y.u\langle y\rangle.Q) :: \Lambda}\,(\text{Cut?})$$

whose reduction rules is: $\nu u.(!u(x).P \mid \nu y.u\langle y\rangle.Q) \longrightarrow \nu u.(!u(x).P \mid \nu y.(Q \mid P\{y/x\}))$.

$$\dfrac{\Gamma; \cdot \vdash P :: x : A \qquad \dfrac{\Gamma, u : A; \Delta, y : A \vdash Q :: \Lambda}{\Gamma, u : A; \Delta \vdash \nu y.u\langle y\rangle.Q :: \Lambda}\text{(CopyR)}}{\Gamma; \Delta \vdash \nu u.(!u(x).P \mid \nu y.u\langle y\rangle.Q) :: \Lambda}\text{(Cut?)}$$

whose reduction rules is: $\nu u.(!u(x).P \mid \nu y.u\langle y\rangle.Q) \longrightarrow \nu u.(!u(x).P \mid \nu y.(P\{y/x\} \mid Q))$. The next are the rules for one and bottom.

$$\dfrac{\Gamma; \cdot \vdash x\langle\rangle.\boldsymbol{\theta} :: x : \mathbf{1} \qquad \dfrac{\Gamma; \Delta \vdash P :: \Lambda}{\Gamma; \Delta, x : \mathbf{1} \vdash x().P :: \Lambda}}{\Gamma; \Delta \vdash \nu x(x\langle\rangle.\boldsymbol{\theta} \mid x().P) :: \Lambda}\text{(CutR)}$$

whose reduction rules is: $\nu x.(x\langle\rangle.\boldsymbol{\theta} \mid x().P) \longrightarrow P$.

$$\dfrac{\dfrac{\Gamma; \Delta \vdash P :: \Lambda}{\Gamma; \Delta, x : 1 \vdash x().P :: \Lambda} \qquad \Gamma; x :\perp \vdash x\langle\rangle.\boldsymbol{\theta} :: \cdot}{\Gamma; \Delta \vdash \nu x(x().P \mid x\langle\rangle.\boldsymbol{\theta}) :: \Lambda}\text{(CutR)}$$

whose reduction rules is: $\nu x(x().P \mid x\langle\rangle.\boldsymbol{\theta}) \longrightarrow P$.

$$\dfrac{\dfrac{\Gamma; \Delta \vdash P :: \Lambda}{\Gamma; \Delta \vdash x().P :: \Lambda, x :\perp} \qquad \Gamma; x :\perp \vdash x\langle\rangle.\boldsymbol{\theta} :: \cdot}{\Gamma; \Delta \vdash \nu x(x().P \mid x\langle\rangle.\boldsymbol{\theta}) :: \Lambda}\text{(CutR)}$$

whose reduction rules is: $\nu x(x().P \mid x\langle\rangle.\boldsymbol{\theta}) \longrightarrow P$.

$$\dfrac{\Gamma; x :\perp \vdash x\langle\rangle.\boldsymbol{\theta} :: \cdot \qquad \dfrac{\Gamma; \Delta \vdash P :: \Lambda}{\Gamma; \Delta, x : \mathbf{1} \vdash x().P :: \Lambda}}{\Gamma; \Delta \vdash \nu x(x\langle\rangle.\boldsymbol{\theta} \mid x().P) :: \Lambda}\text{(CutR)}$$

whose reduction rules is: $\nu x.(x\langle\rangle.\boldsymbol{\theta} \mid x().P) \longrightarrow P$. The next reduction is for the multiplicative conjunction.

$$\dfrac{\dfrac{\Gamma; \Delta_1 \vdash P :: \Lambda_1, y : A \qquad \Gamma; \Delta_2 \vdash Q :: \Lambda_2, x : B}{\Gamma; \Delta_1, \Delta_2 \vdash \nu y.x\langle y\rangle.(P \mid Q) :: \Lambda_1, \Lambda_2, x : A \otimes B}(\otimes R) \qquad \dfrac{\Gamma; \Delta, z : A, x : B \vdash R :: \Lambda}{\Gamma; \Delta, x : A \otimes B \vdash x(z).R :: \Lambda}(\otimes L)}{\Gamma; \Delta, \Delta_1, \Delta_2 \vdash \nu x.(\nu y.x\langle y\rangle.(P \mid Q) \mid x(z).R) :: \Lambda, \Lambda_1, \Lambda_2}\text{(CutR)}$$

whose reduction rules is: $\nu x.(\nu y.x\langle y\rangle(P \mid Q) \mid x(z).R) \longrightarrow \nu x.(Q \mid \nu y.(P \mid R\{y/z\}))$. For the case of the multiplicative disjunction the reduction is:

$$\dfrac{\dfrac{\Gamma;\Delta \vdash P :: \Lambda, x : B, z : A}{\Gamma;\Delta \vdash x(z).P :: \Lambda, x : A \parr B}\,(\parr R) \quad \dfrac{\Gamma;\Delta_1, y : A \vdash Q :: \Lambda_1 \qquad \Gamma;\Delta_2, x : B \vdash R :: \Lambda_2}{\Gamma;\Delta_1,\Delta_2, x : A \parr B \vdash \nu y.x\langle y\rangle.(Q \mid R) :: \Lambda_1,\Lambda_2}\,(\parr L)}{\Gamma;\Delta,\Delta_1,\Delta_2 \vdash \nu x.(x(z).P \mid \nu y.x\langle y\rangle.(Q \mid R)) :: \Lambda,\Lambda_1,\Lambda_2}\,(\text{CutR})$$

whose reduction rules is: $\nu x.(x(z).P \mid \nu y.x\langle y\rangle.(Q \mid R)) \longrightarrow \nu y.(\nu x.(P\{y/z\} \mid R) \mid Q)$. Finally, the replicate and WhyNot rules do not associate with any reduction rule. It is just a renaming.

$$\dfrac{\dfrac{\Gamma;\cdot \vdash P :: y : A}{\Gamma;\cdot \vdash !x(y).P :: x :!A}\,(!\text{R}) \quad \dfrac{\Gamma, u : A;\Delta \vdash Q :: \Lambda}{\Gamma;\Delta, x :!A \vdash Q\{x/u\} :: \Lambda}\,(!\text{L})}{\nu x.(!x(y).P \mid Q\{x/u\})}\,(\text{CutR})$$

whose reduction rules is: $\nu x.(!x(y).P \mid Q\{x/u\}) \longrightarrow \nu u.(!u(y).P \mid Q)$.

A.2 Coq Implementation

This Appendix presents some results within Coq. As we discussed in the Introduction, the mechanization may involve implementation details and technical lemmas that need an explanation. Nevertheless, including these details within the work can be troublesome due to its extension. Thus, this Appendix presents and gives a brief discussion of them.

From now on, the signature in Coq introduces the propositions, and its proof will be a sketch. The following is an example of what the reader should expect for definitions or functions:

```
(**
 Definition for the closed functions.
*)
Definition Close_Name ( k z: nat )( N : Name ) : Name :=
match N with
  | FName n0 ⇒ if ( n0 =? z ) then (BName k) else N
  | BName i ⇒ N
end.
#[global]
Hint Resolve Close_Name : Piull.

Fixpoint Close_Rec (k z : nat)( T : Process ) {struct T} : Process
```

And for lemmas, propositions, or theorems:

Lemma A.1. Exchange Open

```
Lemma Exchange_Open : forall (P : Prepro)(i j x y : nat),
i ≠ j → ({i → x}({j → y} P) = {j → y}({i → x}P)).
```

Proof. Induction over P. □

The idea is that the reader who wants to work out all the details can accomplish this task by reading the Coq code and following the sketch. The first implementation detail is the structure of the implementation. The following are the files in the repository and their contents:

- Defs_*. Them have definitions for the suffix structure. For example, Defs Propositions has the definitions for the propositions.
- Facts_*. Them have basic or routinary results for the suffix structure. For example, Facts FVars has facts relating to free names.

- Props_*. Them have advance and important results for the structure in the suffix. For example, Props FVars has key results relating to free names.

The source code can be found at: https://github.com/cigarcial/Tmcod

A.2.1 Database and Tactics

A custom database helps to take advantage of Coq's automatic proof search. We implement it using the Coq's documentation and recommendations. The name of the database for the project is Piull; in this database, using the following lines, we add any of the constructors or results.

```
Create HintDb Piull.
...
#[global]
Hint Constructors lc_name : Piull.
...
#[global]
Hint Resolve lc_name : Piull.
```

To append results into the database the global keyword is preferred. As part of the automatic proof search, one can define custom tactics. The first solve goals that are disjunctions, searching within all the possibilities

```
(**
  Searching a proof where the goal contains multiple or operators.
  Keep in mind that this tactics is exponential.
*)
Ltac OrSearch :=
  (progress auto with *) +
  (left; OrSearch) +
  (right; OrSearch).
```

The next tactic solves easy inductions over a given structure.

```
(**
  This resolves easy structural inductions over a given term (T).
*)
Ltac StructuralInduction T :=
  intros;
  induction T;
  simpl;
  repeat match goal with
      | IHT : _ |- _ => rewrite IHT
  end;
```

```
  auto with Piull.
```

Lastly, we implemented a tactic that solves the goal using structural induction over the processes and a given lemma for the names.

```
(**
*)
Ltac InductionProcess P Name_Lemma :=
  induction P;
    intros; simpl;
    repeat rewrite Name_Lemma;
    repeat match goal with
      | IHP  : _ |- _  ⇒ rewrite IHP; auto
      | IHP1 : _ |- _ ⇒ try rewrite IHP1; auto
    end.
```

There are some other tactics in Defs Tactics that are not here since are variations of those already shown.

A.2.2 Propositions in Coq

The propositions of linear logic work with an Inductive definition, except for the linear implication. We implement the linear implication using the dual.

```
Inductive Proposition : Type :=
  | ONE : Proposition
  | ABS : Proposition
  | TEN (A : Proposition) (B : Proposition) : Proposition
  | PAR (A : Proposition) (B : Proposition) : Proposition
  | EXP (A : Proposition) : Proposition
  | MOD (A : Proposition) : Proposition.
...
Definition ULLT_IMP (A : Proposition) (B : Proposition) : Proposition := (
     A⊥ ⅋ B).
```

The definition of duals uses a Fixpoint operator.

```
Fixpoint Dual_Prop ( T : Proposition ) : Proposition :=
match T with
...
```

All the results for the propositions follow using induction over the structure of the term or the πULL database. For example,

Lemma A.2. Dual is idempotent

```
Proposition Doble_Duality_ULLT :
```

```
forall A : Proposition ,
(A⊥)⊥ = A.
```

Proof. Induction over A. □

A.2.3 Names

In the case of the names, there is a detail about the implementation:

```
(**
*)
Inductive Name : Type :=
  | FName ( x : nat) : Name
  | BName ( i : nat) : Name.
```

Observe that the bound names use the datatype *nat*, but we used Strings in the presentation. This implementation detail does not conflict with the previous work. Here we prefer the *nat* datatype due to the libraries already implemented in Coq. Any data type that supports boolean and syntax comparations will work.

Previously, most of the Process results follow by induction on the structure and a particular lemma for the names. The following are crucial lemmas of this kind.

Lemma A.3. Lemmas for Names

```
(**
*)
Lemma Eq_Open_Name :
forall ( i y k x p : nat),
i ≠k →
Open_Name i y (Open_Name k x (BName p)) = Open_Name k x (Open_Name i y (
    BName p)).
...
(**
*)
Lemma Subst_Name_Open_Name_Ex :
forall ( x : Name )( x0 y0 z w k: nat ),
FName w = Subst_Name x0 y0 (FName z) →
Subst_Name x0 y0 (Open_Name k z x) = Open_Name k w (Subst_Name x0 y0 x).
...
(**
*)
Lemma Lca_Zero_Lc_Name :
```

```
forall ( x : Name ),
lca_name 0 x ↔ lc_name x.
...
(**
*)
Lemma Subst_Name_Gen_Output :
forall (u x0 : nat)(x : Name),
u ≠ x0 →
u ∈ FVars_Name (Subst_Name u x0 x) → False.
```

Proof. We analyse the structure of the name (bound, free) and the cases for the variable within the term (equal, non-equal). □

Some of the lemmas preserve the same proof structure, therefore we use special tactics to help solving these cases.

```
(**
*)
Ltac DecidSimple x y :=
  destruct (bool_dec (x =? y) true);
  match goal with
    | e : (x =? y ) = true |- _ ⇒
      (rewrite e; apply beq_nat_true in e; rewrite e; progress auto with
          Piull) +
      (apply beq_nat_true in e; lia; progress auto with Piull) +
      (try rewrite e)
    | n : (x =? y ) ≠ true |- _ ⇒
      (apply not_true_iff_false in n; try rewrite n; progress auto with
          Piull) +
      (apply not_true_iff_false in n; try apply beq_nat_false in n; try
          contradiction; progress auto with Piull) +
      (apply not_true_iff_false in n)
  end;
  auto with Piull.
...
(**
*)
Ltac DecidEq :=
  match goal with
    | Gt : ?num0 = ?num1 |- _ ⇒ apply beq_nat_true_inv in Gt; rewrite Gt
  end;
  auto with Piull.
```

A.2.4 Free Names in Coq

To implement the set of free names in Coq, we used the Coq Ensamble implementation and its libraries.

```
(**
*)
Definition FVarsE := Ensemble nat.
```

Then, the implementation of the free name set is as expected:

```
(**
  Free names for a given term.
*)
Definition FVars_Name ( N : Name ) : FVarsE :=
match N with
  | FName x ⇒ Singleton nat x
  | BName i ⇒ Empty_set nat
end.

Fixpoint FVars ( T : Process ) {struct T} : FVarsE :=
match T with
  | Pzero ⇒ Empty_set nat
  | Fuse x y ⇒ (FVars_Name x) ∪(FVars_Name y)
  ...
  | Chan_output x y P ⇒ (FVars_Name x) ∪(FVars_Name y) ∪(FVars P)
  ...
```

Given the distinct operations on processes, it is relevant to know how to change the free names set when one of these operations is applied.

Lemma A.4. Operations and Free Names

```
(**
*)
Lemma FVars_Open_Beq :
forall ( P : Process)(u x i: nat),
u ≠x → ( u ∈ FVars P ↔ u ∈ FVars ({i → x}P)).
...
(**
*)
Lemma FVars_Open :
forall (Q : Process)( y x i : nat),
x ∈ FVars ( {i → y} Q ) → x = y ∨ x ∈ FVars ( Q ).
...
(**
*)
```

```
Lemma FVars_Beq_Close :
forall ( Q : Process)(x x0 i : nat),
x ≠x0 → x ∈ FVars (Close_Rec i x0 Q) →
x ∈ FVars (Q).
...
(**
*)
Lemma FVars_Close_NotIn :
forall ( P : Process )( x x0 i: nat),
x ≠x0 → ¬ x ∈ FVars (Close_Rec i x0 P) → ¬ x ∈ FVars (P).
...
(**
*)
Lemma FVars_Subst :
forall ( P : Process )( x y x0 : nat ),
x ∈ FVars ({y \ x0} P) → x = y ∨ x ∈ FVars (P).
```

Proof. Induction over the process, unfolding, and analyzing the output of the operation. □

A.2.5 Processes in Coq

The processes implementation uses an inductive definition.

```
Inductive Process : Type :=
  | Pzero : Process
  | Fuse (x y : Name) : Process
  | Parallel (P Q : Process ) : Process
  | Chan_output ( x y : Name ) (P : Process) : Process
  | Chan_zero (x : Name ) : Process
  | Chan_close ( x : Name ) ( P : Process ) : Process
  (* Processes with bounded names *)
  | Chan_res (P : Process ) : Process
  | Chan_input ( x : Name ) (P : Process) : Process
  | Chan_replicate ( x : Name)(P : Process ) : Process.
#[global]
Hint Constructors Process : Piull.
```

One of the relevant differences in the implementation is that the notations are different. Table A.2.5 shows the translation between Definition 2.4 and the Coq notation.

In the Facts file, there are relevant results concerning processes. The design of the results is in such a way that it requires the minimum amount of hypotheses.

LNR	$\to$	Coq
$\mathbf{0}$		$\mathbf{0}$
$[N \leftrightarrow N]$		$[N \leftrightarrow N]$
$P \mid Q$		$P \downarrow Q$
$\nu.P$		νP
$N\langle N\rangle.P$		$N\langle N\rangle P$
$N(_).P$		$N \cdot P$
$N().P$		$N() \cdot P$
$!N(_).P$		$N! \cdot P$
$N\langle\rangle.\mathbf{0}$		$N \cdot \mathbf{0}$

```
(**
*)
Lemma Lca_Process_Rd :
forall ( P : Process )( k x: nat ),
lca (S k) P → lca k ({k → x} P).

(**
*)
Lemma Lca_Open_Close_Subst :
forall ( P : Process )( x y k : nat ),
lca k P → { k → y } Close_Rec k x P = { y \ x } P.
```

The proof of the results, in most of the cases, follows by induction on the structure of the process or the *lc* and *lca* predicates. Two important results concerning the process are:

```
(**
*)
Theorem Congruence_WD :
forall P Q : Process,
(P \cong Q) → lc(P) → lc(Q).

(**
*)
Theorem ProcessReduction_WD :
forall P Q : Process,
(P ⟶ Q) → lc(P) → lc(Q).
```

A.2.6 LN Representation in Coq.

The definitions and results for the LN representation are in the files of the processes. The proofs using the *lca* and *lc* predicates have two parts, the

part for names, and the part that relies on the process' structure.

```
(** lc
*)
Inductive lc_name : Name → Prop :=
  | lc_fname : forall (x : nat), lc_name (FName x).
#[global]
Hint Constructors lc_name : Piull.

Inductive lc : Process → Prop :=
...
  | lc_chan_close : forall (x : Name)(P : Process),
    lc_name x → lc P → lc ( x ()· P )

  | lc_chan_res : forall (P : Process),
    ( forall (x : nat), lc ({ 0 → x }P) ) → lc (ν P)
...

(** lca
*)
Inductive lca_name : nat → Name → Prop :=
  | lca_Fname : forall ( k x : nat), lca_name k (FName x)
  | lca_Bname : forall ( k i : nat), ( i < k ) → lca_name k (BName i).
#[global]
Hint Constructors lca_name : Piull.

Inductive lca : nat → Process → Prop :=
...
  | lca_parallel : forall ( k : nat )( P Q : Process ),
    lca k P → lca k Q → lca k (P | Q)

  | lca_chan_output : forall ( k : nat )( x y : Name )( P : Process ),
    lca_name k x → lca_name k y → lca k P → lca k ( x \langle y \rangle ·
      P)
...
```

The multiple open operations are considered a part of the representation, hence they are in the same file of definitions. But, there is a different file for the results concerning these operations: Facts_MOpen.

```
(**
*)
Lemma MOpen_Name_BName_Gt :
forall (k i : nat)(L : list nat),
(length L) = k → k <= i →
```

```
MOpen_Name_Rec k L (BName i) = (BName i).

(**
*)
Theorem Lca_Lc_Process_MOpen :
forall (P : Process)(k : nat)(L : list nat),
(length L) = k →
lca k P → lc (MOpen_Rec k L P).
```

Most of the results follow by induction on the list.

```
(**
*)
Lemma Subst_Lc_Lc :
forall (P : Process)(x y : nat),
lc P → lc ({y \ x} P).

(**
*)
Lemma Body_Lc_One :
forall ( P : Process ),
Body P → lca 1 P.

(**
*)
Lemma Subst_Lca_Process :
forall ( P : Process )( k : nat ),
lca k P → forall (x y : nat ), lca k ({y \ x} P).
```

A.2.7 Types

The implementation of the types is split into several files. The first includes the definitions for the contexts.

```

Inductive Assignment : Type := assig ( x : Name )( A : Proposition ) :
    Assignment.
Notation " x : A " := (assig x A )(at level 60).
#[global]
Hint Constructors Assignment : Piull.

Definition Context := Ensemble Assignment.
```

The manipulation of contexts requires auxiliary definitions like Collect, Bld, SMA, Replace, etc; for which there are technical results.

```
Lemma SMA_Collect :
forall (G : Context)(x : nat)(A : Proposition),
Collect G → Collect (SMA G x A).

Lemma SMA_Union_Bld :
forall ( G : Context )( x : nat )( A : Proposition),
SMA (Bld x A ∪ G) x A = G .
```

Note that some of the discussed details in Section 3.5 are relevant in this part of the implementation. The second part is the definition of the typing rules. To do a correct implementation, we require appropriate conditions.

```
(**
*)
Inductive Good_Contexts : Context → Context → Context → Process →
    Prop := is_well_collected :
forall (D F G : Context)(P : Process),
  ( forall (x : nat), ( (x ∈ FVars P) → ( exists (A : Proposition), (FName x
      :A) ∈ (D ∪ F ∪ G) ) )) ∧
  ( (Disjoint_Sets D F) ∧ (Disjoint_Sets D G) ∧ (Disjoint_Sets F G) ∧ (
      Injective D) ∧ (Injective F) ∧ (Injective G))
   → (Good_Contexts D F G P).
#[global]
Hint Constructors Good_Contexts : Piull.

(**
*)
Reserved Notation "D ';;;' F '!-' P ':::' G" (at level 60).
Inductive Inference : Process → Context → Context → Context → Prop :=
...
  | otiml : forall ( D F G: Context )( x y : nat )( y : nat )( A B :
      Proposition )( P : Process ),
    Collect D → Collect F → Collect G → lc P → x ≠y →
    Good_Contexts D ( (Bld x B) ∪ (Bld y A) ∪ F ) G P →
    Good_Contexts D ( (Bld x (A ◦ times B)) ∪ F ) G (FName x · Close y P) →
    ( D ;;;  ( (Bld x B) ∪ (Bld y A) ∪ F ) !− P :::  G ) →
    ( D ;;;  ( (Bld x (A ◦ times B)) ∪ F ) !− (FName x · Close y P) :::  G )
...
```

The results concerning interactions between the typing rules and the LN representation are in the Facts files.

```
Lemma FVars_Reduction :
forall ( P Q : Process )( x : nat),
```

```
x ∈ FVars P → P ⟶ Q → x ∈ FVars Q.

(**
*)
Lemma No_Typing_Fuse_One_Lf :
forall ( A : Proposition )( x y : nat )( D F G : Context ),
( (FName x : A) ∈ D ) → \not ( D ;;; F !− ([FName x \leftrightarrow FName y
    ]) ::: G ).

(**
*)
Lemma TS_GContext_Type_Subst_Lf :
forall ( x y : nat )( A : Proposition )( D F G : Context )( P : Process ),
Fresh y (D ∪ F ∪ G) → x ≠y → lc P →
Good_Contexts D ((Bld x A) ∪ F) G P →
Good_Contexts D ((Bld y A) ∪ F) G (Close x P ⊥y).
```

The implementation finishes with substitution lemmas, transference lemmas, and the Subject Reduction Theorem. They are in the Props file.

```
(**
*)
Lemma Transference_Rg_Lf :
forall ( P : Process)(D F G : Context),
D;;;  F !− P :::  G →
forall (x : nat)(A : Proposition), ( (FName x : A) ∈ G ) →
D;;;  (F ∪ Bld x (A ⊥\perp )) !− P :::  (SMA G x A).

(**
*)
Theorem Soundness :
forall (P : Process)(D F G : Context),
  ( D ;;;  F !− P :::  G ) → forall (Q : Process), (P ⟶ Q) → ( D ;;; F !− Q :::
      G ).
```

Bibliography

[Bar96] Andrew G. Barber. Dual intuitionistic linear logic. Technical report, 1996.

[Ber36] Paul Bernays. Alonzo church. an unsolvable problem of elementary number theory. american journal of mathematics, vol. 58 (1936), pp. 345–363. *Journal of Symbolic Logic*, 1(2):73–74, 1936.

[BU07] Stefan Berghofer and Christian Urban. A head-to-head comparison of de bruijn indices and names. *Electronic Notes in Theoretical Computer Science*, 174(5):53–67, 2007. Proceedings of the First International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2006).

[Cha12] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning - JAR*, 49:1–46, 10 2012.

[CPT12] Luís Caires, Frank Pfenning, and Bernardo Toninho. Towards concurrent type theory. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 01 2012.

[de 72] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[DLdVMY19] Ugo Dal Lago, Marc de Visme, Damiano Mazza, and Akira Yoshimizu. Intersection types and runtime errors in the pi-calculus. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[Gir87] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, January 1987.

[Gir93] Jean-Yves Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59(3):201–217, 1993.

[Gir95] Jean-Yves Girard. Linear logic: Its syntax and semantics. In *Proceedings of the Workshop on Advances in Linear Logic*, page 1–42, USA, 1995. Cambridge University Press.

[GL86] J.Y. Girard and Y. Lafont. Linear logic and lazy computation. Technical Report RR-0588, INRIA, December 1986.

[GM94] Bierman Gavin M. *On Intuicionist Linear Logic*. PhD thesis, University of Cambridge, UK, 1994.

[Hon93] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, pages 509–523, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[Kam01] Fairouz Kamareddine. Reviewing the Classical and the de Bruijn Notation for λ-calculus and Pure Type Systems. *Journal of Logic and Computation*, 11(3):363–394, 06 2001.

[Kle36] S. C. Kleene. λ -definability and recursiveness. *Duke Math. J.*, 2(2):340–353, 06 1936.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, Heidelberg, 1982.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, USA, 1999.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40, 1992.

[Pai90] Valeria Correa Vaz De Paiva. *The Dialectica Categories*. PhD thesis, University of Cambridge, UK, 1990.

[Pfe95] Frank Pfenning. Structural cut elimination. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*, LICS '95, page 156, USA, 1995. IEEE Computer Society.

[SU06] M.H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Number v. 10 in Lectures on the Curry-Howard isomorphism. Elsevier, 2006.

[SW03] D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2003.

[Ton15] Bernardo Toninho. A logical foundation for session-based concurrent computation. 2015.

[Tro91] Anne Sjerp Troelstra. *Lectures on Linear Logic*. Center for the Study of Language and Information Publications, 1991.

[vdHP20] Bas van den Heuvel and Jorge A. Pérez. Session type systems based on linear logic: Classical versus intuitionistic. *Electronic Proceedings in Theoretical Computer Science*, 314:1–11, Apr 2020.

[vP13] J. von Plato. *Elements of Logical Reasoning*. Cambridge University Press, 2013.

[Wad12a] Philip Wadler. Propositions as sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, page 273–286, New York, NY, USA, 2012. Association for Computing Machinery.

[Wad12b] Philip Wadler. Propositions as sessions. *SIGPLAN Not.*, 47(9):273–286, September 2012.