UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JAQUELINE KLEIN GALLI

π-Calculus Semantic with Values Passing in the Denotational Approach

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Daltro José Nunes

Porto Alegre
August 2020

CIP — CATALOGING-IN-PUBLICATION

Galli, Jaqueline Klein

π-Calculus Semantic with Values Passing in the Denotational Approach / Jaqueline Klein Galli. – Porto Alegre: PPGC da UFRGS, 2020.

61 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2020. Advisor: Daltro José Nunes.

1. Formal Methods. 2. Pi-Calculus. 3. Denotational Semantic. 4. Values Passing. I. Nunes, Daltro José. II. Título.

"If I have seen further than others,
it is because I stood on the shoulders of giants."
— Sir Isaac Newton

ACKNOWLEDGEMENTS

My sincere gratitude to my family and friends, who always believed in me, and supported me in my choices. A special thank you to my dad Paulo for your companionship, patience and emotional support at all times, my mom Vilma for your love and motivation, and my brother Gustavo for never let me give up.

I would especially like to thank my advisor Prof. Dr. Daltro José Nunes for his guidance through the development of this dissertation, for believing in me when no one else did, and for never treating me differently because of gender, age, or area of study. Also, to thank for always being available to meet my difficulties with dedication and patience, and for serving as a reference of the researcher I want to be. Without his support and motivation, this work would not be possible.

I would also like to thank my friends and colleagues from the Federal University of Rio Grande do Sul (UFRGS), for their help, support, friendship, ideas and companionship. To those whose name I did not mention, but helped me developing this work in any way, thank you all. Finally, I must acknowledge that this research would not have been possible without the financial assistance of CNPq and the infrastructure provided by the Federal University of Rio Grande do Sul.

ABSTRACT

In the last recent years, there was a noticeable increase in the use of formal verification by the industry, and the reason for that is because the formal languages are each time more enhanced, and the tools that support those languages were developed. Although there is a higher use in critical systems, the application can also avoid execution errors and, consequently, update the system with bugs' correction as a goal. The extension of these languages makes them wider in their applicability and the creation of tools makes the use easier for lay or lack of experience users. The use of formal languages in a critical systems development ensures the operation to work and, at the same time, the benefit-cost of the time used in a formal verification of a system specification, is lower when a tool is available. The formal π-*Calculus* method is largely used in labeled transition and mobile systems, because it shows competition, mobility and creation of new links. Even with a wide applicability, it presents some limitations for the user, for example, there is neither formal language nor tools that can support it. π-*Calculus* is an extension of CCS (Calculus of Communication System) that shows operation semantics, however, no value passing. In this dissertation, is presented a denotational semantic for the π-*Calculus*, based in the mapping of syntactic phrases in semantics domain through semantics function, that gives structure for a creation of a formal language. Besides that, the denotational approach makes the tools implementation much easier, as it is the case of LOTOS and Maude. Along the denotational approach, we introduce the concepts of local memory (storage), temporary memory (environment) and types of environment, providing the ideal conditions so is possible to demonstrate how the π-*Calculus* values passing occur. Thereby it was possible to verify, in an objective way, the mobility in this method, once that is clear how the communication channels passing works, peculiarity not found at CCS.

Keywords: Formal Methods. Pi-Calculus. Denotational Semantic. Values Passing.

Semântica do π-Cálculo com Passagem de Valores na Abordagem Denotacional

RESUMO

Nos últimos anos nota-se um aumento significativo na utilização de verificação formal pela indústria, pois as linguagens formais estão cada vez mais aprimoradas e ferramentas que suportem essas linguagens foram desenvolvidas. Embora tenha maior utilização em sistemas críticos, a sua aplicação também pode evitar erros de execução e, consequentemente, a atualização de sistemas com propósito de correções de bugs. As extensões dessas linguagens as tornam mais abrangentes em sua aplicabilidade e a criação de ferramentas faz com que seu uso seja mais fácil para usuários leigos ou pouco experientes no assunto. O uso de linguagens formais no desenvolvimento de sistemas críticos assegura o funcionamento do sistema e, ao mesmo tempo, o custo/benefício do tempo usado na verificação formal de uma especificação de um sistema é menor quando se tem uma ferramenta. O método formal π-*Cálculo* é muito usado em sistemas de transição rotulada e sistemas móveis, pois apresenta concorrência, mobilidade e criação de novos links. Por mais abrangente que seja sua aplicabilidade, apresenta algumas limitações ao usuário como, por exemplo, não possui uma linguagem formal e nem ferramenta que o suporte. O π-*Cálculo* é uma extensão do CCS (Cálculo de Sistemas de Comunicação) que apresenta semântica operacional, entretanto sem passagens de valores. Nesta dissertação, apresentamos uma semântica denotacional para o π-*Cálculo* baseada no mapeamento de frases sintáticas em domínios semânticos através de funções semânticas, que dá estrutura para a criação de uma linguagem formal. Além disso, a abordagem denotacional facilita muito a implementação de ferramentas, como no caso do LOTOS e do Maude. Juntamente com a abordagem denotacional introduzimos os conceitos de memória (local), ambiente (temporária) e ambiente de tipos, proporcionando as condições necessárias para que pudéssemos demonstrar como se dá a passagem de valores no π-*Cálculo*. Assim foi possível verificar de forma objetiva a mobilidade nesse método uma vez que fica claro como se dá a passagem de canais de comunicação, particularidade não encontrada no CCS.

Palavras-chave: Métodos Formais, Pi-Cálculo, Semântica Denotacional, Passagem de Valores.

LIST OF ABBREVIATIONS AND ACRONYMS

BNF	Backus-Naur Form
CCS	Calculus of Communicating Systems
CSP	Communicating Sequential Processes
ISO	International Organization for Standardization
LOTOS	Language of Temporal Ordering Specification
MWB	Mobility Work-bench
OSI	Open Systems Interconnection
PNs	Petri Nets

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Formal languages, and mathematical based methods, have been created to contain characteristics and formal semantics to develop critical systems. Cohen, Harwood and Jackson (1986) classify formal methods into three different types: concurrent methods, algebraic methods, and model-based methods; but also many of them have extensions that allow them to develop systems in other areas. In this dissertation, the concern is precisely about the concurrent π-*Calculus* method (MILNER; PARROW; WALKER, 1992).

The π-*Calculus* method is an extension of another concurrent method, called CCS (MILNER, 1984), importing its structure and operations, such as parallelism and synchronism. However, it is better that the CCS method in some aspects; for example, it adds the ability to create new communication channels; therefore, it facilitates mobility, parameter dependence and dynamic reconfiguration. The most significant applicability of the π-*Calculus* method ranges from modeling telecommunications protocols to modeling object-oriented languages.

The usual semantic base used for the construction of this kind of method is the operational semantics, adopted by Milner (1999), that uses inference rules. All of its extensions, and based literature, use this same semantics, which facilitates mathematical proofs of the processes evolution through reaction rules. However, it has been quite difficult to mature a formal language for this method, which consequently creates tools for systems development use. Only the Pict programming language by Pierce and Turner (2000) is based on π-*Calculus*, but it is still at the experimental stage.

Primarily for those reasons, this study presents the development of a denotational semantics for an extended syntax of π-*Calculus*, based on an already elaborated arrangement, by Milner (1999) and Sangiorgi and Walker (2003). The first one is presented as *Basic* π-*Calculus*, it contains the most basic syntax to define the behavior of a process, and the second, as *Typed* π-*Calculus*, is an extension, which presents a type of system for the method.

Semantics are responsible for representing the meaning of each syntactic phrase. In denotational semantics, this meaning is specified by a denotational value for some domain. Once the domain D and a class of phrases P are defined, a semantic function is introduced, which maps each phrase in P to its D denotation: $f : P \longrightarrow D$, that is, it maps a context-free language to mathematical entities (WATT, 1996).

Afterward, the concept of local memory (store) and temporary memory (environment), which didn't exist in the π-*Calculus* literature yet, will be presented. The concept of memory will be based on the idea presented by Milner (1984) in CCS, so that there is not much distinction from the existing literature. The environment idea does not exist yet, for both CCS and π-*Calculus* methods, and, therefore, it will be based on the syntax created in this dissertation, and also on the denotational semantics presented by Watt (1996). This model is suitable for storing a wide variety of imperative languages that have types of values stored in locations.

Programs written with imperative languages run easier on hardware, and because of this, this type of software has become prevalent. When this language is composed with declarative languages, they become more powerful and able to access generic computational resources, as long as this integration does not affect the fundamental principles of the declarative language (BARENDREGT; MANZONETTO; PLASMEIJER, 2013). For a method to be efficient, in the construction of a language that fulfills this demand, it is essential to understand the behavior of agents in passing arguments. These arguments are usually called values, which in the case of the π-*Calculus* method has the same construction of variables and channels.

Since the goal of this study is to show the passing of values in π-*Calculus*, these arguments must be inserted in a type of environment, once the values, variables, and channels of communication are labeled in the same form. Therefore, a denotational semantics must be made for *Typed* π-*Calculus* to demonstrate how it is possible for both to pass value expansions and mobility, which is one of the main characteristics of this method. With that, the passing of values is the last part of this dissertation.

1.1 Related Works

1.1.1 Values Passing in CCS

In CCS, the passing of values between agents happens through gates, channels of communication, in which an observable experiment sends a name through a positive port, and receives it over a negative port, to variables free of context. This relationship is represented by $P \xrightarrow{\alpha} P'$, where the agent P admits an α-*experiment* and can turn into P' as the result.

For information on CCS with values passing, see Milner (1984). For values passing in the π-*Calculus*, we shall use *Typed* π-*Calculus* (SANGIORGI; WALKER, 2003) to identify the active types in agents in observable experiments, to be able to pass between ports, such as in

LOTOS.

1.1.2 Values Passing in LOTOS

The algebraic formal language *LOTOS*, introduced by Bolognesi and Brinksma (1987), is a formal description technique developed within ISO for the formal specification of open distributed systems, and, in particular, for those related to the Open Systems Interconnection (OSI) computer network architecture.

In *Basic LOTOS*, observable actions are identified only by the name of the gate where they are offered, and processes can only have a finite number of gates. In *Full LOTOS* the action is constituted by three components: a gate, a list of events, and an optional predicate.

An event can either offer(!) or accept(?) a value. Predicates serve to establish a condition on the values that can be accepted/offered. The processes synchronize their actions, as long as they name gates with the same name, the event lists are combined, and the predicates, if any, are satisfied (LAI; JIRACHIEFPATTANA, 2012).

According to Logrippo, Faci and Haj-Hussein (1992), synchronization between processes with value exchange, occurs when two or more processes agree on a single tuple of values, and established in a gate. This is the case of corresponding actions. More precisely, if a set of processes $P_1, ..., P_n$ is composed in parallel, and G represents the list of common ports, on which synchronization must occur, these processes can synchronize if the following conditions are valid:

1. $P_1, ..., P_n$ are all offering actions $a_1, ..., a_n$, respectively;

2. The gates of action $g_1, ..., g_n \in G$ are the same;

3. $T = \mathit{tuples}(a_1) \frown \cdots \frown \mathit{tuples}(a_n) \neq \varnothing$, where $\frown$ is a concatenation operator.

Process outputs are denoted by $E_1, ..., E_n$, where $E_1, ..., E_n$ is a set of value expressions and results of the process execution. These results are passed on to another activated behavior process if allowed (synchronism).

1.2 Outline

The remainder of this dissertation is organized as follows. In chapter 2, the π-*Calculus* method is presented, and its aspect is described as a concurrent method. In chapter 3 a denotational semantics is made for the *Basic* π-*Calculus* syntax, based on a syntax presented by Milner (1999). The following chapter 4 shows a denotational semantics for the *Typed* π-*Calculus* syntax, based on a syntax presented by Sangiorgi and Walker (2003). In chapter 5 according to the syntax and semantics made, it is showed how to pass the values in the π-*Calculus* method. Lastly, chapter 6 presents the conclusion and detailed future studies.

2 THE METHOD

This chapter is intended for a brief presentation of the π-*Calculus* method and the characteristics of a concurrent method.

2.1 Concurrence

Concurrent methods address specifications for systems that involve automata behavior and exhibit parallelism, concurrency, and communication in their components, interacting at any time during their executions, with the system having to manage simultaneous communication between them. These components are called agents or processes that behave dynamically in parallel in an internally sequential manner, communicating on a more abstract or more stable time scale, based on timeless state transitions (BUSTARD, 1990).

A concurrent system composed of agents or sequential processes that communicate with each other, behaves involving the analysis of four factors: the internal behavior of each agent, their external behavior, the structure of the system into agents and the communication between them (COHEN; HARWOOD; JACKSON, 1986).

The internal behavior of an agent depends on the structure of its internal state, the set of initial states, the set of operations that can access these states, the accessible states, and the possible state transitions. Given the initial states and the operations sets, the internal specification implies a set of possible sequences of state transitions.

Since the state transition does not meet internal operations, an alternative is to consider that these operations will be performed as a result of an agent's interaction with other agents. Thus, the agent describes behavior restrictions as limitations on the acceptable sequence of events. The specification of this behavior, in the form of predicates on the events, in which the agent can participate, is called the specification of external behavior, that is, the event has externally observable behavior.

Communication is the mechanism through which events are transmitted between agents. In the event of a simultaneous occurrence, between the events of each agent, the competing system will depend on the internal state transitions. Examples of concurrent methods are CCS (MILNER, 1984), π-Calculus (MILNER, 1999), CSP (ABDALLAH, 2005) and PNs (PETERSON, 1977).

2.2 π-Calculus Method

The formal π-*Calculus* method is aimed at analyzing communication ownership between concurrent processes, and interacting among concurrent systems. Mobility, where new links can be created and broken between existing components, is the name given for this kind of evolution. In addition, it is possible to model this mobility between components.

This method is oriented to competing systems, aimed at the mobility of communication channels. As an extension of CCS, it imports all its functions: composition, restriction, substitution, and synchronism, adding other functions that support the transfer of communication channels between agents. In π-*Calculus*, there is no difference between names, values, and variables in the transition between agents, so when a channel is transmitted, all labels restricted to it must be renamed for future interaction with other agents (MILNER, 1999).

The base for an agreement between operations comes from the structural congruence. This means that, there are different ways to write two processes, but they will have the same meaning. The definition of structural congruence for π-*Calculus* has already been elaborated by Milner (1999), which can be found in the literature, so there will be only references to it in this study. It is through this equivalence relationship that the reaction rules were made.

The input and output primitives of this method are monadic: exactly one label or channel is exchanged during each communication. The polyadic π-*Calculus* is a useful extension that allows the atomic communication of tuples of labels/channels, which in this dissertation, will be called: messages. Also, it has the polymorphic systems type by allowing generic operations, i.e., operations which can be safely applied to many different types of argument, like list operations. (TURNER, 1995).

For better understanding, this method will be divided into two parts: *Basic* π-*Calculus*, which involves the basic concepts developed by Milner (1999), and *Typed* π-*Calculus*, an extension of the method that involves the concepts of Type Systems developed by Sangiorgi and Walker (2003).

3 BASIC π-CALCULUS

Basic π-*Calculus* uses concurrence and mobility for the passing of the values, variables, and links. In this environment, there is no distinction between them and, therefore, they are just called labels or names. In the same way, the processes can carry unrestricted names as long as values are the result of reduced terms and only as long as there is synchronism.

The reduction of terms occurs in a similar way to algebraic languages, such as Maude (NUNES, 2012). At this time, there is still concern about the type of data being transported, but how and if the system can send it or receive it. To define the grammar, we apply a BNF[1] in order to simplify the semantic specifications, and the denotacinal semantic is based in Watt (1996) and Slormeger and Kurtz (1995).

3.1 Label

In π-*Calculus*, we assume an infinite and countable set of names $a,\ b,\ x,\ y,\ ...$, representing the links (channels), variables, and calculation values. We define the set of names as follows:

$$Name = \{a, b, x, y, \cdots\} \tag{3.1}$$

Given the set *Name*, we can introduce a new set $\overline{Name}$ defined as:

$$\overline{Name} \overset{def}{=} \{\overline{n} \mid n \in Name\} \tag{3.2}$$

where the elements are called *co-names*. We assume that *Name* and $\overline{Name}$ are disjoint sets and we denote *Label* as the union of these sets. (MILNER, 1999)

Definition 1. *(Label) Let Label* $\overset{def}{=} Name \cup \overline{Name}$ *be a set of labels, where each element of Label is used for labelled transition systems and ports. The ports are used to connect systems that, when synchronized, can exchange messages. For all* $x \in Label$*, we call "*$\overline{x}$*" the* ***complement*** *of "*x*".*

[1]"When applied to programming languages, this notation is known as Backus-Naur Form or BNF for the researchers who first used it to describe Algol60. Note that BNF is a language for defining languages—that is, BNF is a metalanguage." (SLORMEGER; KURTZ, 1995)

3.1.1 Label: Semantic Functions and Equations

Assuming that the sets *Name* and $\overline{Name}$ are disjoint, one can get a bijective function $f = \overline{}$, defined by Milner (1984) a bijection from *names* to *co-names*:

$$\overline{} \;:\; Name \longrightarrow \overline{Name} \tag{3.3}$$

where

$$\forall\, n \in Name : \overline{n} \in \overline{Name}. \tag{3.4}$$

In addiction, the inverse bijective function $f^{-1} = \overline{}$, can be obtained as follows:

$$\overline{} \;:\; \overline{Name} \longrightarrow Name,\ where\ \forall\, \overline{n} \in \overline{Name} : \overline{\overline{n}} = n. \tag{3.5}$$

Thus, the function $f = f^{-1} = \overline{}$ is a bijection over *Label*. In particular for any subset $L_1 \subset Label$, the set $\overline{L_1} = \{\overline{x} \mid x \in L_1\}$ satisfies $\overline{L_1} \subset Label.$

In some cases it will be important to consider the function *name* over *Label* whose codomain is *Name*:

$$name : Label \longrightarrow Name \tag{3.6}$$

where if $\forall\, x, \overline{x} \in Label$, and $\overline{x}$ is **complement** of x then:

$$name(x) = name(\overline{x}) = x. \tag{3.7}$$

Also, we extend to sets $L : Label$ by defining:

$$names(L) = \{name(x);\ x \in L\}. \tag{3.8}$$

One can identify some properties for *names*. Let $L_1, L_2 \subset Label$ be subsets, as follows:

$$names(L_1) = names(\overline{L_1}) \tag{3.9}$$

$$names(L_1 \cup L_2) = \{name(x);\ x \in L_1\} \cup \{name(y);\ y \in L_2\} \tag{3.10}$$

$$names(L_1 \cap \overline{L_1}) = names(L_1) \tag{3.11}$$

$$names(L_1 \cap \overline{L_2}) = names(L_1) \cap names(\overline{L_2}) \tag{3.12}$$

$$names(L_1 \times L_2) = \{name(x);\ x \in L_1\} \cup \{name(y);\ y \in L_2\} \tag{3.13}$$

3.2 Message

To carry a message, it is necessary that one link can send and receive more than one name. It means that a message can be an empty tuple, a unit (monadic Calculus) or a homogeneous tuple (polyadic Calculus) of labels. By definition, a message is defined as follows:

$$Message = \{\langle\, x_1\ \cdots\ x_n \,\rangle \mid x_1, ..., x_n \in Label, n \in \mathbb{N}\} \cup \{\langle\ \rangle\}, \tag{3.14}$$

where we will assueme that $\langle\ \rangle$ is a empty message. By notation, we will use $\vec{x}$ when the tuple is $\langle\, x_1\ \cdots\ x_n \,\rangle$.

3.2.1 Semantic Domains

As a denotational semantics is being developed for reductions in *π-Calculus*, it can be considered that the equation reduction renders respect the sets already instantiated as sorts with the declaration of their operators in any formal language. Therefore, it was used the sorts already instantiated in the formal language Maude, for reasons of familiarity of this dissertation's authors. Since the Label and Message sets are not instantiated in Maude, these sets' instantiating follows as sorts.

According to the message definition, we can specify the sort Message as a tuple of labels. In this case, first follows the specification of sort Label:

```
fmod LABEL is
***Declaration of sort***
sort Label .
***Declaration of operators***
ops a ā b b̄ ··· z z̄ ··· :  -> Label [ctor] .
```

```
op _=_ : Label Label -> Bool .
op _≠_ : Label Label -> Bool .
***Declaration of variables***
vars x1 x2: Label .
***Declaration of equations***
ceq x1 = x2 = true if x1==x2 .
ceq x1 = x2 = false if x1=/=x2 .
ceq x1 ≠ x2 = false if x1==x2 .
ceq x1 ≠ x2 = true if x1=/=x2 .
endfm
```

Now that the sort *Label* was well defined, one can specify *Message*. The first structure is a theory as follows:

```
fth TH is
sort Component .
endfth
```

The second structure is the instantiation of a tuple of size *n* as follows:

```
fmod TUPLES { X :: TH } is
sorts Tuple N-Tuple .
subsort X$Component < N-Tuple .
op <> : -> Tuple .
op _ _ : X$Component N-Tuple -> N-Tuple .
op <_> : N-Tuple -> Tuple .
op < _ _ > : X$Component Tuple -> Tuple .
vars c1 c2 : X$Component .
var t : N-Tuple .
eq < c1 <> > = < c1 > .
eq < c1 < c2 t > > = < c1 c2 t > .
endfm
```

The third is a *view* that assigns the sort Component to the sort *Label* for parameterization of the tuple in MESSAGE:

```
view TH-as-LABEL
from TH to LABEL is
sort Component to Label .
endv
```

The sort *Message* is defined by instantiation of sort *Tuple* by sort *Label*, as follows:

```
fmod MESSAGE is
protecting TUPLES { TH-as-LABEL } * (sort Tuple to Message) .
endfm
```

3.2.2 Message: Semantic Functions and Equations

The function *names* defined to *Message* generates a set of all names on Label enclosed in the message. In that way, the set of names of a message $M : Message$, with $L : Label$, is defined as follows:

$$names(M) = names(\underbrace{L \times L \times \cdots \times L}_{n}) : L^n \subseteq M, n \in \mathbb{N}. \qquad (3.15)$$

The equations as follows:

$$names(\langle\ \rangle) = \varnothing \qquad \textit{empty message} \quad (3.16)$$

$$names(\langle\ x\ \rangle) = \{name(x) \mid x \in L\} \qquad \textit{monadic} \quad (3.17)$$

$$names(\langle\ x_1\ x_2\ \cdots\ x_n\ \rangle) = \{name(x_i) \mid x_i \in L,\ i = 1, ..., n\} \qquad \textit{polyadic} \quad (3.18)$$

3.3 Link

When labels, e.g. x and $\bar{y}$, are used to label transitions, it means that one process is interacting with another process through these labels. In this case, these labels are called *links* (ports). A link interacts with its **complement** by sending or receiving *messages*. Therefore a link can be:

- **output**, sends a message when linked with its complement;
- **input**, receives a message when linked with its complement.

This way it can classify a link by a label function for its state:

$$Link = Label \longrightarrow \mathbf{output} + \mathbf{input} \tag{3.19}$$

Let $x, \bar{x} \in Label$ be labels, the equations as follows:

$$Link(\bar{x}) = \mathbf{output} \tag{3.20}$$

$$Link(x) = \mathbf{input} \tag{3.21}$$

In π-*Calculus*, the names of communicating channels don't lose their labels after synchronised. So, the function *relabelling* (defined to CCS) is not be used in π-*Calculus*. Willing to do a internal behaviour of a process is used the operator "**new** x", with $x \in Label$, to restrict a link for exchange data.

3.3.1 Synchronism

A synchronism happens when there is a pair of the complementary labels $(x, \bar{x})$. This pair is responsible for the creation a communicating channel where a message can be sent or received. In this case, each one these labels is a *link*.

In π-Calculus, a synchronism is represented by the character "$\boldsymbol{\tau}$" and has the highest precedence on the process. It means that the synchronism is a silent action and occurs regardless of the interaction with the environment. It is possible to define the properties of a link through the function:

$$\mathit{Synchronism} = \mathit{Link} \times \mathit{Link} \longrightarrow \boldsymbol{\tau} + \textbf{fail} \tag{3.22}$$

Let $x, y \in \mathit{Label}$ be labels, the equation as follows:

$$\mathit{Synchronism}(x, y) = \mathit{if}\ (x\ \mathit{is}\ \textbf{complementary}\ \mathit{to}\ y)\ \mathit{then}\ \boldsymbol{\tau}\ \mathit{else}\ \textbf{fail} \tag{3.23}$$

3.4 Conditional

The π-*Calculus* can be extended to the restricted condition: *match*. Also, it can use more than one condition and require that the calculation meets all the conditions or fail if they do not. To give meaning to the match, it is defined in the specification of the LABEL sort.

The syntax of *Conditional* is of the form where the labels can be the same (match) or a conditional followed by another conditional successively. Let $x_1, x_2 \in \mathit{Label}$, the BNF as follows:

$$\mathit{Conditional}\ ::=\ [\, x_1 = x_2 \,]\ \big|\ \mathit{Conditional}\ \mathit{Conditional} \tag{3.24}$$

3.4.1 Semantic Domains

Adding *Conditional* in a process, the language becomes richer. However, it is necessary to consider domains that aid in the understanding of the syntax. Again, let us use a public domain: *Bool*. A truth precondition ensures that the process can occur and a false one, stops the process.

3.4.2 Conditional: Semantic Functions and Equations

The function *condition* is defined from *Conditional* to *Bool*:

$$\mathit{condition} : \mathit{Conditional} \longrightarrow \mathit{Bool} \tag{3.25}$$

For this function we have the following equations, let $x_1, x_2 \in \mathit{Label}$ and $\mathit{cnd}_1, \mathit{cnd}_2 \in \mathit{Conditional}$:

$$\mathit{condition}[\![\ [\ x_1 = x_2\ ]\ ]\!] \ = \ \mathit{if}\ (x_1 = x_2)\ \mathit{then\ true\ else\ false} \tag{3.26}$$

$$\mathit{condition}[\![\mathit{cnd}_1\ \mathit{cnd}_2]\!] = \mathit{condition}(\mathit{cnd}_1) \wedge \mathit{condition}(\mathit{cnd}_2) \tag{3.27}$$

3.5 π Prefix

A prefix on the process that will be defined later describes the experiment that will occur before the process begins. Let $x, \bar{x} \in \mathit{Label}$ and $y, z \in \mathit{Message}$, the syntax to prefix π is defined as:

$$\pi \ ::= \ \bar{x}\, y \mid x\, z \mid \boldsymbol{\tau} \mid \mathit{Conditional}\ \pi \tag{3.28}$$

According Sangiorgi and Walker (2003), the prefix definition is as follows:

Definition 2. *(Prefix π) The prefix π has a π firing sequence that must occur before a process. Let $x \in$ Label, $y, z \in$ Message, the informal semantic as follows:*

- *output prefix: $\bar{x}\, y$ - It means that the message y can be sends via link x;*
- *input prefix: $x\, z$ - It means that the z can receives a message via link x. In this case, all occurrences of the z must be replaced by the message received.*
- *synchronise prefix: $\boldsymbol{\tau}$ - It means that a process occurs without external interference, expressing an internal action of a process.*
- *match prefix: $[\ x_1 = x_2\ ]\ \pi$ - It means that the prefix π fires if the x_1 and x_2 are the same labels.*

3.5.1 π Prefix: Semantic Functions and Equations

Now, it one can to define a function from π to a codomain formed by an ordered pair (Link, Message) when a message is being sent or received, τ when there is synchronism, or **fail** if the prefix fails.

$$pref : \pi \longrightarrow (Link \times Message) + \boldsymbol{\tau} + \textbf{fail} \tag{3.29}$$

Let $x, \bar{x} \in Label$, $y, z \in Message$, $\boldsymbol{\tau}$ is a synchronism and $cnd \in Conditional$, the equations as follows:

$$pref[\![\, \bar{x}\, y \,]\!] = (Link(\bar{x}), y) \tag{3.30}$$

$$pref[\![\, x\, z \,]\!] = (Link(x), z) \tag{3.31}$$

$$pref[\![\, \boldsymbol{\tau} \,]\!] = \boldsymbol{\tau} \tag{3.32}$$

$$pref[\![\, cnd\ \pi]\!] = if\ condition(cnd)\ then\ pref(\pi)\ else\ \textbf{fail} \tag{3.33}$$

3.6 Process

A process in π-*Calculus* is a linked sequence of prefixes which guarantees that the exchange of the messages can only happen after the synchronism between links of the same name. Also, there is the possibility of mobility when processes receive and send links to create new connections.

To ensure the mobility, it is necessary to define *Process* by a set of the satisfactory operations. In this case, are define the language π-*Basic*(**nill**, **.**, $+$, $|$, **new**, **Replicate**) with terms given as follows:

$$Summation ::= \textbf{nill} \;\big|\; \pi\ \textbf{.}\ Process \;\big|\; Summation + Summation \tag{3.34}$$

$$Process ::= Summation \;\big|\; Process \mid Process \;\big|\; \textbf{new}\ x(\ Process\ ) \;\big|\; \textbf{Replicate}\ Process \tag{3.35}$$

$$\textbf{Replicate}\ Process ::= Process \;\big|\; \textbf{Replicate}\ Process \mid Process \tag{3.36}$$

According Sangiorgi and Walker (2003), the process definition is as follows:

Definition 3. *(Process) Let $S, S' \in Summation$, $P \in Process$ and $x \in Label$. So, the informal semantic as follows:*

- *inaction: the process **nill** represents an empty sum and that there is nothing that can be done.*

- *prefixed process:* π **.** *P is a prefixed process, where* π *is a prefix, and P just can be occur after* π *has been satisfied. The symbol "***.***" (below) represents a sequential definition.*
- *sums:* $S + S'$ *means that the capabilities of both processes are available and just one will be choosen. The other will be lost. This is a non-deterministic choice, and the agent can act like S or S'.*
- *composition:* $P \mid P'$ *means that the processes can proceed independently and interact using shared links.*
- *restriction:* ***new*** $x\ (P)$ *means that the x, with* $x \in Name$*, is restricted to internal behavior of process P.*
- *replication:* ***Replicate*** *P means a infinite sequence of the composition of the process P with itself.*

To understand the behavior of the prefixed process, it is necessary firs to define an auxiliary domain *Action*. The action has a function to describe what is happens with the process in a determined time.

3.6.1 Action

A *labelled transition* is defined by a set of the actions applied to the prefixed process transition. Actions are used to labeled transition relationships of the process. It means that transitions occur according to prefixed actions, which consequently determine the behavior of the processes according to your structure. It one define the actions like an auxiliary set that do not form part of the syntax.

Let $x, \bar{x} \in Label$ and $y, z \in Message$, the set of actions is defined as:

$$Action\ ::=\ \bar{x}\,y \mid x\,y \mid \bar{x}\,(z) \mid \tau \tag{3.37}$$

Definition 4. *(Action) The labelled transition relations are labelled by actions. Let* $n \in Name$ $x, \bar{x} \in Label$ *and* $y, z \in Message$*, the informal semantic as follows:*

- *free output:* $\bar{x}\,y$ *- It means that the process sends y via x;*
- *input:* $x\,y$ *- It means that the process receives y via x;*

- *bound output: $\bar{x}\ (z)$ - It means that a process send a local name z, i.e., z is a restricted name;*
- *synchronism: $\boldsymbol{\tau}$ - It means that a process occurs without external interference, expressing a internal synchronized action.*

3.6.1.1 Action: Semantic Functions and Equations

To describe the behavior of action, it defines the function *act* from *Action* to a codomain with the elements ordered pair (*Link* × *Message*), where *Link* is responsible for sending or receiving a free or restrict *Message* and $\boldsymbol{\tau}$, when the action is silent, as follows:

$$act : Action \longrightarrow (Link \times Message) + \boldsymbol{\tau} \tag{3.38}$$

Let $x,\ \bar{x},\ z \in Label$, $y \in Message$, $\boldsymbol{\tau}$ is a synchronism and $cnd \in Conditional$, the equations to *act* are as follows:

$$act[\![\bar{x}\ y]\!] = (Link(\bar{x}), y) \tag{3.39}$$

$$act[\![x\ y]\!] = (Link(x), y) \tag{3.40}$$

$$act[\![\bar{x}\ (z)]\!] = (Link(\bar{x}), \textbf{new}\ z) \tag{3.41}$$

$$act[\![\boldsymbol{\tau}]\!] = \boldsymbol{\tau} \tag{3.42}$$

3.6.1.2 Substitution

The interaction between a process and action can bring forth a *substitution*. It means that a set of identifiers can be associated with values and links through the communicating channels.

In π-*Calculus*, a *Message* can be a value, a variable or a link. For understanding we will call *Identifier* Message elements set that define a variable. In π-*Basic*, it can be represented as a Message subsort.

$$Identifier < Message \tag{3.43}$$

The substitution function *subs* is applied over *Identifier* to *Message*.

$$subs : Identifier \longrightarrow Message \tag{3.44}$$

Definition 5. *(Substitution) Let Substitution be the set of all substitution functions, the finite mapping from Identifier to Message, where Identifier is (by definition) a subset of variables in Message. We write* $\{\vec{v}/\vec{id}\}$*, with* $\vec{id} \in$ *Identifier and* $\vec{v} \in$ *Message, such that* $subs(id_j) = v_j$ *for all* $id_j \in \vec{id}$ *and* $v_j \in \vec{v}$*.*

The poliadic π-*Calculus*, defined as in Milner (1999), ensures that a substitution can be occur like in monadic π-Calculus provided the lengths of message and identifier be equal, it means, the substitution $\{\vec{v}/\vec{id}\}$ occur, if, and only if, $\mid \vec{v} \mid = \mid \vec{id} \mid$.

In π-*Calculus* there is two types of binding labels: when a label is a restriction to process (e.g. **new** *n P*, *n* is bounded), or when all occurrences of a label on a process can be substituted to another label received via a link (e.g. $x\,z\,.\,P$, z is bounded). The definition by Sangiorgi and Walker (2003) is as follows:

Definition 6. *(Binding) A label is called binding in a process when it is bonded to the process scope. The binding labels set of a process P is represented by bn(P). Any label not bound to a process P is called free label and is an element of fn(P). Let* $n, x, \bar{x} \in$ *Label,* $y, z \in$ *Message and* $P \in$ *Process, follows:*

$- bn(\textbf{nill}) = \varnothing$	$- fn(\textbf{nill}) = \varnothing$
$- bn(\bar{x}\ y\,.\,P) = bn(P)$	$- fn(\bar{x}\,y\,.\,P) = \{x, y\} \cup fn(P)$
$- bn(x\,z\,.\,P) = \{z\} \cup bn(P)$	$- fn(x\,z\,.\,P) = \{x\} \cup (fn(P) \setminus \{z\})$
$- bn(\boldsymbol{\tau}\,.\,P) = bn(P)$	$- fn(\boldsymbol{\tau}\,.\,P) = fn(P)$
$- bn([\,n_1 = n_2\,]\,P) = bn(P)$	$- fn([\,n_1 = n_2\,]\,P) = \{n_1, n_2\} \cup fn(P)$
$- bn(P_1 + P_2) = bn(P_1) \cup bn(P_2)$	$- fn(P_1 + P_2) = fn(P_1) \cup fn(P_2)$
$- bn(P_1 \mid P_2) = bn(P_1) \cup bn(P_2)$	$- fn(P_1 \mid P_2) = fn(P_1) \cup fn(P_2)$
$- bn(\textbf{new}\ n\,.\,P) = \{n\} \cup bn(P)$	$- fn(\textbf{new}\ n\,.\,P) = fn(P) \setminus \{n\}$
$- bn(\textbf{Replicate}\ P) = bn(P)$	$- fn(\textbf{Replicate}\ P) = fn(P)$

3.6.1.3 Transition Relations

The labeled transition relations are defined over prefixed processes by a second-order function:

$$actpref : \pi\,.\,Process \to (Action \to Process + \textbf{abort}) \tag{3.45}$$

Let $\alpha \in$ *Action*, $x,\ \bar{x},\ y_2 \in$ *Label*, $y_1,\ z \in$ *Message* and $P \in$ *Process*, the equations as follows:

$$actpref[\![\bar{x}\, y_1 \,.\, P]\!]\ (\bar{x}\, y_1) = P \tag{3.46}$$

$$actpref[\![\bar{x}\, y_2 \,.\, P]\!]\ (\bar{x}\, (y_2)) = \textbf{new}\ y_2\ (actpref(\bar{x}\, y_2 \,.\, P)\ (\bar{x}\, y_2)) \tag{3.47}$$

$$actpref[\![x\, z \,.\, P]\!]\ (x\, y) = \{{}^{y}\!/\!{}_{z}\}P \tag{3.48}$$

$$actpref[\![\boldsymbol{\tau} \,.\, P]\!]\ (\boldsymbol{\tau}) = P \tag{3.49}$$

$$actpref[\![[x = x]\pi \,.\, P]\!]\ \alpha = actpref(\pi \,.\, P)\ \alpha \tag{3.50}$$

To all the other combinations between prefix and action, the result is **abort**. Because this the equations was omitted and will be treated as a deadlock.

3.6.2 Process: Semantic Functions and Equations

According to the syntax, three functions are necessary to denote the behavior of the expressions defined as follows:

$$sum : Summation \longrightarrow (Action \longrightarrow \textbf{nill} + \textbf{abort}) \tag{3.51}$$

$$trans : Process \longrightarrow Summation \tag{3.52}$$

$$reply : \textbf{Replicate}\ Process \longrightarrow Process \tag{3.53}$$

Let $\alpha \in Action$, $S, S' \in Summation$ and $P, P' \in Process$, the equations as follows:

$$sum[\![\ \textbf{nill}\ ]\!]\ \alpha = \textbf{nill} \tag{3.54}$$

$$sum[\![\ \pi \,.\, P\ ]\!]\ \alpha = \begin{cases} if\ actpref(\pi \,.\, P\ \alpha) \neq \textbf{abort} \\ then\ trans(actpref(\pi \,.\, P\ \alpha)) \\ else\ \textbf{abort} \end{cases} \tag{3.55}$$

$$sum[\![\ S + S'\ ]\!]\ \alpha = if\ sum(S)\ \alpha \neq \textbf{abort}\ then\ sum(S)\ \alpha\ else\ sum(S')\ \alpha \tag{3.56}$$

$$trans[\![\ S\ ]\!] = if\ sum(S) \neq \textbf{abort}\ then\ sum(S)\ else\ \textbf{nill} \tag{3.57}$$

$$trans[\![\ P \mid P'\ ]\!] = trans(P) \mid trans(P') \tag{3.58}$$

$$trans[\![\ \mathbf{new}\ x\ P\ ]\!] = \mathbf{new}\ x\ (trans(P)) \tag{3.59}$$

$$trans[\![\ \mathbf{Replicate}\ P\ ]\!] = reply(\mathbf{Replicate}\ P) \tag{3.60}$$

$$reply[\![\ P\ ]\!] = trans(P) \tag{3.61}$$

$$reply[\![\ \mathbf{Replicate}\ P \mid P\ ]\!] = reply(\mathbf{Replicate}\ P) \mid trans(P) \tag{3.62}$$

Example 1. *Reduces* $P = (\bar{x}\,y\,.\,Q \mid x\,z\,.\,R)$*:*

$$trans(\bar{x}\,y\,.\,Q \mid x\,z\,.\,R) = trans(\bar{x}\,y\,.\,Q) \mid trans(x\,z\,.\,R) \tag{3.63}$$

$$= sum(\bar{x}\,y\,.\,Q)\ (\bar{x}\,y) \mid sum(x\,z\,.\,R)\ (x\,y) \tag{3.64}$$

$$= trans(actpref(\bar{x}\,y\,.\,Q)\ (\bar{x}\,y)) \mid trans(actpref(x\,z\,.\,R)\ (x\,y)) \tag{3.65}$$

$$= trans(Q) \mid trans(\{{}^{y}\!/_{z}\}R) \tag{3.66}$$

3.7 The Process Reduction

The *prefix* and *trans* functions can define the reaction relations of a prefix π and a *process*, respectively. However, to understand what happens in a process reduction, these functions are not enough. To achieve this, some semantic domains as storage and environment are necessary, where the prefix can find a possible action during a process reduction.

Below is an example of an informal reduction of process *P*, using functions built is this work and how happens the behavior of the names in the storage and environment.

Example 2 (Process Behaviour)**.** *Let* $P, Q, R \in$ *Process and* $x, y, z \in$ *Name, reduces* $P = (\bar{x}\,y\,.\,Q \mid x\,z\,.\,R)$*:*

Table 3.1: Reduction process of the *P* with environment behaviour and memory state (storage).

Process	Environment	Memory
$trans(P) = trans(\bar{x}\,y\,.\,Q \mid x\,z\,.\,R)$	$\mathbf{empty}\epsilon$	$loc\ x \mapsto unused$
$= trans(\bar{x}\,y\,.\,Q) \mid trans(x\,z\,.\,R)$	$\mathbf{empty}\epsilon$	$loc\ x \mapsto unused$
$= sum(\bar{x}\,y\,.\,Q)\ (\bar{x}\,y) \mid sum(x\,z\,.\,R)\ (x\,y)$	$\mathbf{empty}\epsilon$	$loc\ x \mapsto unused$
$= trans(actpref(\bar{x}\,y\,.\,Q)\ (\bar{x}\,y)) \mid trans(actpref(x\,z\,.\,R)\ (x\,y))$	$var\ z \mapsto loc\ x$	$loc\ x \mapsto undefined$
$= trans(Q) \mid trans(\{{}^{y}\!/_{z}\}R')$	$var\ z \mapsto loc\ x$	$loc\ x \mapsto val\ y$

In the end, it one have $trans(\overline{x}\, y \,.\, Q \mid x\, z \,.\, R) \overset{*}{\longrightarrow} trans(Q) \mid trans(\{{}^{y}/_{z}\}R')$ *and the location* x *with identifier* z *is allocated with the value* y*.*

3.7.1 Memory

The storage concept for CCS has been defined by Milner (1984), and it will be extended for π-*Calculus* in this section. First, it is necessary to know about the *register* of reading of the value and writing of the variable behavior to one or more processes. Let $y \in Message$ be the value, $z \in Message$ be the variable and $x \in Label$ be the link, the register $R(x)[y] \in Process$ is defined as follows:

$$R(x)[y] \overset{def}{=} \overline{x}\, y \,.\, R(\overline{x})[y] + x\, z \,.\, R(x)[z],\ \textit{where } z \in Message. \tag{3.67}$$

It means that there exist two options to $sum(R(x)[y])$ according to prefix $\overline{x}\, y$ or $x\, z$ chosen:

- $trans(actpref(\overline{x}\, y \,.\, R(\overline{x})[y])\ (\overline{x}\, y)) = R(\overline{x})[y]$, means that a value y was identified over link $\overline{x}$ and thus the result is the *register* of reading the y.
- $trans(actpref(x\, z \,.\, R(x)[z])\ (x\, y)) = trans(\{{}^{y}/_{z}\}R(x)[z]) = R(x)[y]$, means that a value y was mapped to variable z over link $\overline{x}$, and thus the result is the *register* of writing the y.

By these means, the location process (storage position), $L(x)[z] \in Process$, is defined as follows:

$$L(x)[z] \overset{def}{=} x\, z \,.\, R(x)[z], \tag{3.68}$$

where $z \in Message$ is the name of a location in which a value can be stored, and $x \in Label$ is a communication channel between the location and possible value that can be allocated to it. This location can be:

- $L(x)[\ ] =$ **unused** (simply $L(x)$), i.e., it was not allocated to any message.
- $L(x)[z] =$ **undefined**, i.e., is not yet known the value allocated in z.
- $L(x)[{}^{y}/_{z}] = R(x)[y]$, means that z was allocated with a message through link x.

Thus, it one has the sets, with $i \in \{1, ..., n\}$ and $n \in \mathbb{N}^*$:

- $Writable = \bigcup L(x_i)[z_i]$, formed for all writable locations $L(x_i)$ associated with variables z_i respectively, of a process, where z_i are all distinct messages and x_i are all distinct links;

- *Memory* $= \{L(x_1)[z_1],\ \cdots,\ L(x_n)[z_n]\}$, the storage, where each register cannot communicate with each other;
- *Readable* $= \bigcup R(\overline{x_i})[y_i]$, formed by all readable messages y_i.

From this moment, the storage will be called like your usual definition of memory. To update the memory in each transition, the auxiliary function is introduced:

$$\textit{update} : \textit{Memory} \times \textit{Writable} \times \textit{Readable} \longrightarrow \textit{Memory} \tag{3.69}$$

where, let $m \in \textit{Memory}$ be a memory, $\textit{loc} \in \textit{Writable}$ be a writable location and $\textit{val} \in \textit{Readable}$ be a readable message, so:

- *update(m, loc, val) = m'* is the same memory *m*, except that in *m'*, the local *loc* will be allocated by a message of the value *val*.

Now, it one defined the equation that updates this memory:

$$\textit{update}(m, L(x)[z], R(x)[y]) = m[{}^{y}/_{z}] \tag{3.70}$$

3.7.2 Environment

For each process, we define an environment ε that includes mappings to readable messages from all predefined writable locations:

$$\varepsilon = \{(z \mapsto L(x)[\,]) \mid z \in \textit{Identifier and } x \in \textit{Label}\} \tag{3.71}$$

To explain the behavior of the environment at each step of the reduction process, follow the auxiliary functions:

$$\mathbf{empty}\varepsilon : \varepsilon \tag{3.72}$$
$$\textit{findwtble} : \varepsilon \times \textit{Link} \longrightarrow \textit{Writable} \tag{3.73}$$
$$\textit{bind} : \textit{Identifier} \times \textit{Writable} \longrightarrow \varepsilon \tag{3.74}$$
$$\textit{overlay} : \varepsilon \times \varepsilon \longrightarrow \varepsilon \tag{3.75}$$

where, let $\epsilon \in \varepsilon$, $x, y \in \textit{Message}$, $\textit{loc} \in \textit{Writable}$ and $l \in \textit{Link}$, so:

- **empty**ε is the empty environment.
- *findwtble*$(\epsilon, Link(x))$ gives the **unused** location pointed by link x in ϵ environment.
- *bind*$(z, L(x)[\,])$ gives an environment with a simple binding between an identifier z and a location *unused* pointed by communication channel x.
- *overlay*(ϵ, ϵ') gives the combined environment between ϵ' and ϵ, where, if any location was allocated in both ϵ and ϵ' then your value in ϵ' overrides in ϵ. When a environment receives a new mapping from a z identifier to a $L(x)[\,]$ writable, it one defines $\epsilon[L(x)[z]]$.

Let $\epsilon \in \varepsilon$, $x, \overline{x} \in Label$ and $y, z \in Message$, the equations as follows:

$$findwtble(\epsilon, Link(x)) = L(x)[\,] \tag{3.76}$$

$$bind(z, findwtble(\epsilon, Link(x))) = L(x)[z] \tag{3.77}$$

$$overlay(\epsilon, bind(z, findwtble(\epsilon, Link(x)))) = \epsilon[L(x)[z]] \tag{3.78}$$

3.7.3 Reduction of Process

Once the concept of Memory and Environment has concluded, e can rewrite the functions defined before.

$$sum : Summation \longrightarrow (Action \longrightarrow \varepsilon \longrightarrow Memory \longrightarrow (\mathbf{nill} + \mathbf{stop}) \times Memory) \tag{3.79}$$

$$trans : Process \longrightarrow (\varepsilon \longrightarrow Memory \longrightarrow Summation \times \varepsilon \times Memory) \tag{3.80}$$

$$reply : \mathbf{Replication}\ Process \longrightarrow (\varepsilon \longrightarrow Memory \longrightarrow Process \times \varepsilon \times Memory) \tag{3.81}$$

Let $x, \overline{x} \in Label$, $y, z \in Message$, $\alpha \in Action$, $\epsilon \in \varepsilon$, $m \in Memory$, $S, S' \in Summation$ and $P, P' \in Process$, the equations is as follows:

$$sum[\![\ \mathbf{nill}\ ]\!]\ \alpha\ \epsilon\ m = \mathbf{nill}\ m \tag{3.82}$$

$$sum[\![\ \pi\ .\ P\ ]\!]\ \alpha\ \epsilon\ m = \begin{cases} if\ (sum(\pi\ .\ P)\ \alpha \neq \mathbf{abort}) \\ then\ (sum(\pi\ .\ P)\ \alpha\ \epsilon\ m) \\ else\ \mathbf{stop} \end{cases} \tag{3.83}$$

$$sum[\![\overline{x}\ y\ .\ P]\!]\ (\overline{x}\ y)\ \epsilon\ m = trans(actpref(\overline{x}\ y.\ P\ (\overline{x}\ y))\ \epsilon\ update(m, L(x)[\,], R(x)[y]) \tag{3.84}$$

$$sum[\![x\,z\,.\,P]\!]\ (x\,y)\ \epsilon\ m = trans(actpref(x\,z\,.\,P\ (x\,y)))\ \epsilon[L(x)[z]]\ m[{}^{y}\!/\!{}_{z}] \tag{3.85}$$

$$sum[\![\boldsymbol{\tau}\,.\,P]\!]\ \tau\ \epsilon\ m = trans(actpref(\tau\,.\,P\ \tau)\ \epsilon'\ m' \tag{3.86}$$

$$sum[\![[x = x]\pi\,.\,P]\!]\ \alpha\ \epsilon\ m = trans(actpref([x = x]\pi\,.\,P)\ \alpha)\ \epsilon\ m \tag{3.87}$$

$$sum[\![\ S + S'\ ]\!]\ \alpha\ \epsilon\ m = \begin{cases} if\ (sum(S)\ \alpha \neq \textbf{abort}) \\ then\ (trans(S)\ \alpha\ \epsilon\ m) \\ else\ (trans(S')\ \alpha\ \epsilon\ m) \end{cases} \tag{3.88}$$

$$trans[\![\ S\ ]\!]\ \epsilon\ m = (sum(S))\ \epsilon\ m \tag{3.89}$$

$$trans[\![\ P \mid P'\ ]\!]\ \epsilon\ m = (trans(P) \mid trans(P'))\ \epsilon\ m \tag{3.90}$$

$$trans[\![\ \textbf{new}\ x\ P\ ]\!]\ \epsilon\ m = \textbf{new}\ x\ (trans(P)\ \epsilon\ m) \tag{3.91}$$

$$trans[\![\ \textbf{Replicate}\ P\ ]\!]\ \epsilon\ m = reply(\textbf{Replicate}\ P)\ \epsilon\ m \tag{3.92}$$

$$reply[\![\ P\ ]\!]\ \epsilon\ m = (trans(P), \epsilon, m) \tag{3.93}$$

$$reply[\![\ \textbf{Replicate}\ P \mid P\ ]\!]\ \epsilon\ m = (reply(\textbf{Replicate}\ P) \mid trans(P))\ \epsilon\ m \tag{3.94}$$

Example 3 (Process Behaviour)**.** *Reduces $P = (\bar{x}\ y\ .\ Q \mid x\ z\ .\ R)$ of the Example 2 using the equations defined for environment behavior and memory state at each step of the reduction process:*

Table 3.2: Reduction process of the P with environment behaviour and memory state.

Process	Environment	Memory
$trans(P) = trans(\bar{x}\,y\,.\,Q \mid x\,z\,.\,R)$	ϵ	m
$= trans(\bar{x}\,y\,.\,Q) \mid trans(x\,z\,.\,R)$	ϵ	m
$= sum(\bar{x}\,y\,.\,Q)\ (\bar{x}\,y) \mid sum(x\,z\,.\,R)\ (x\,y)$	ϵ	m
$= trans(actpref(\bar{x}\,y\,.\,Q)\ (\bar{x}\,y)) \mid trans(actpref(x\,z\,.\,R)\ (x\,y))$	$overlay(\epsilon, L(x)[z])$	$update(m, L(x)[\,], R(x)[y])$
$= trans(Q) \mid trans(\{{}^{y}\!/\!{}_{z}\}R')$	$\epsilon[L(x)[z]]$	$m[{}^{y}\!/\!{}_{z}]$

In the end, it one have $trans(p)\ \epsilon\ m\ =^{*}\ (trans(Q) \mid trans(\{{}^{y}\!/\!{}_{z}\}R'))\ \epsilon[L(x)[z]]\ m[{}^{y}\!/\!{}_{z}]$ and the location x with identifier z is allocated with the value y.

3.7.4 Structural Congruence

There are cases where the process is syntactically different, but they have the same behavior. It means that structural congruence identifies only a few agents where the syntactic structure makes it obvious that they are the same. To define structural congruence, it is necessary the concept of *context* and *structural congruence*:

$$Context \ ::= \ [\cdot] \mid Context[Process] \tag{3.95}$$

Definition 7. *(Context) Let $\zeta \in$ Context and $P \in$ Process, the informal semantics as follows:*

- *hole ([·]) - It means a non-degenerate occurrence of* ***nill*** *in a term of Process. Example: $[\cdot] \mid P$.*
- *replace ($\zeta[P]$) - It means that ζ is a context where the hole was replaced by P. Example: if $\zeta_0[\cdot] = [\cdot] \mid P$, then $\zeta_0[Q] = Q \mid P$.*

Definition 8. *(Congruence) An equivalence relation S on processes is a congruence if $(P, Q) \in S$ implies $(\zeta[P], \zeta[Q]) \in S$ for every context ζ and process P and Q. (SANGIORGI; WALKER, 2003)*

Now, it is possible to define a structural congruence by Milner (1999):

Definition 9. *(structural congruence) Two process expressions P and Q in the π-Calculus are structurally congruent, written $P \equiv Q$, if we can transform one into the other by using the following equations:*

- *change of bound names;*
- *reordering of terms in a summation;*
- *identity: $P \mid$* ***nill*** *$\equiv P$;*
- *commutative: $P \mid Q \equiv Q \mid P$;*
- *associative: $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$;*
- ***new*** *$x\ (P \mid Q) \equiv P \mid$* ***new*** *$x\ Q$ if $x \notin fn(P)$;*
- ***new*** *x* ***nill*** *$\equiv$* ***nill****;*

- ***new*** *xy P* $\equiv$ ***new*** *yx P;*

- ***Replicate*** *P* $\equiv$ *P* | ***Replicate*** *P.*

The semantic function is defined from *Context* to *Process* as follows:

$$replace : Context \longrightarrow Process \tag{3.96}$$

Let $[\cdot], \zeta \in Context$ and $P \in Process$, then:

$$replace([\cdot]) = \textbf{nill} \tag{3.97}$$

$$replace(\zeta[P]) = for\ \forall\ [\cdot] \in \zeta\ do\ [\cdot] = P \tag{3.98}$$

3.8 A Basic π-Calculus Application

This section will be present an application of the *basic π-Calculus* semantics in an example. At this time, it one created a semantics to work with *π-Calculus* synchronism, so the problem chosen is the *Dining Chinese Philosophers Problem.* This problem is a generalization of the classical Dining Philosophers Problem by Dijkstra (1978).

3.8.1 Dining Chinese Philosophers Problem

Three philosophers are seated around a circular dining table. Each philosopher has a pasta dish to eat. Besides that, they have sashis in place of forks, and each needs two sashis to eat. The initial state of dinner disposes of one sashi between a pair of pasta dishes. The sashi must be shared with adjacent philosophers. The problem proposes that all philosophers should eat using two adjoining sashis, alternated between eating and thinking.

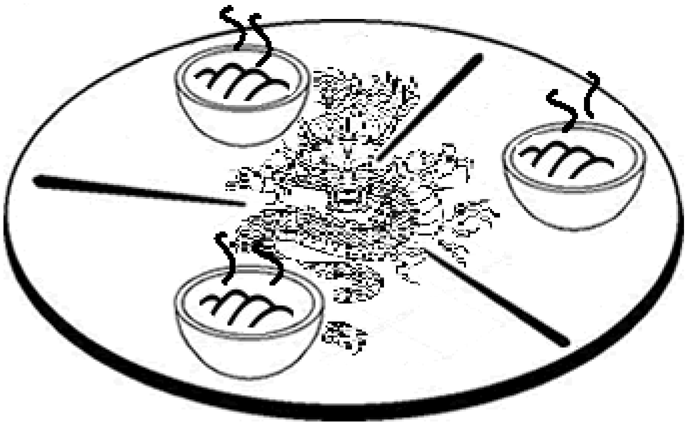
There are a few rules that one can make about the problem:

- If the philosopher has one hashi, then he can receive another and stay with two, or he can send his hashi to another adjacent philosopher.

- If the philosopher runs out of hashi, then he can receive one hashi, or two hashis from adjacent philosophers (seated left or right) one of each.

- If the philosopher has two hashis, then he eats and sends two hashis to adjacent philosophers (seated left and right) to each.

Figure 3.1: Initial state of the Dining Chinese Philosophers.



To study the behavior of the system Dining for $n \in \mathbb{N}$, was been defined the following sets:

- $HASHI = \{hashi_0, hashi_1, hashi_2, hashi_3, ..., hashi_n\}$, where $hashi_0$ means empty-hashi (no hashi) with $HASHI \subset Message$ a subset of the *hashi values* over *Message*. Also, $IDHASHI = \{hashi_r, hashi_l\}$, with $IDHASHI \subset Message$, the set of the *hashi identifiers* over *Message*.

- $PHILOSOPHER = \{Philosopher_1, Philosopher_2, ..., Philosopher_n\}$, is a subset of the philosopher process over *Process*, $PHILOSOPHER \subset Process$.

- $PATH = \{right, left, \overline{right}, \overline{left}\}$, a set of the communication channels, with $PATH \subset Label$.

Let $i, j = 1, ..., n$, with $n \in \mathbb{N}$, the process that represents the behavior of each philosopher is expressed as follows:

$$Philo_{k\in\{i,j\}}[i,j] \overset{def}{=} Philosopher_i[hashi_i, hashi_{j_1}] + Philosopher_j[hashi_{j_2}, hashi_j],$$

where

$$j_1 = \begin{cases} (i+1), & if\ i \neq n. \\ 1, & if\ i = n. \end{cases} \quad and \quad j_2 = \begin{cases} (j-1), & if\ 1 < j \leq n. \\ n, & if\ j = 1. \end{cases}$$

Since a philosopher can send (*Send*) or receive (*Receive*) hashis, the auxiliary processes and the behavior of each philosopher were defined as follows:

- $Send(\overline{right}, \overline{left})[hashi_i, hashi_j] \overset{def}{=} (\overline{right}\ \langle\ hashi_i\ \rangle\ \ |\ \ \overline{left}\ \langle\ hashi_j\ \rangle)$ **.** $Philosopher_{\{i,j\}}$ $[hashi_0, hashi_0]$, it means that the philosopher only can send one hashi to the right or left, or both, to adjacent philosophers if he has more than one.

- $Receive(right, left)[hashi_l, hashi_r] \overset{def}{=}$ **new** *right left* $(right\ (hashi_r)\ |\ left\ (hashi_l))$ **.** $Philosopher_k[hashi_l, hashi_r]$, it means that the philosopher can receive one hashi from each adjacent philosophers to his right or left.

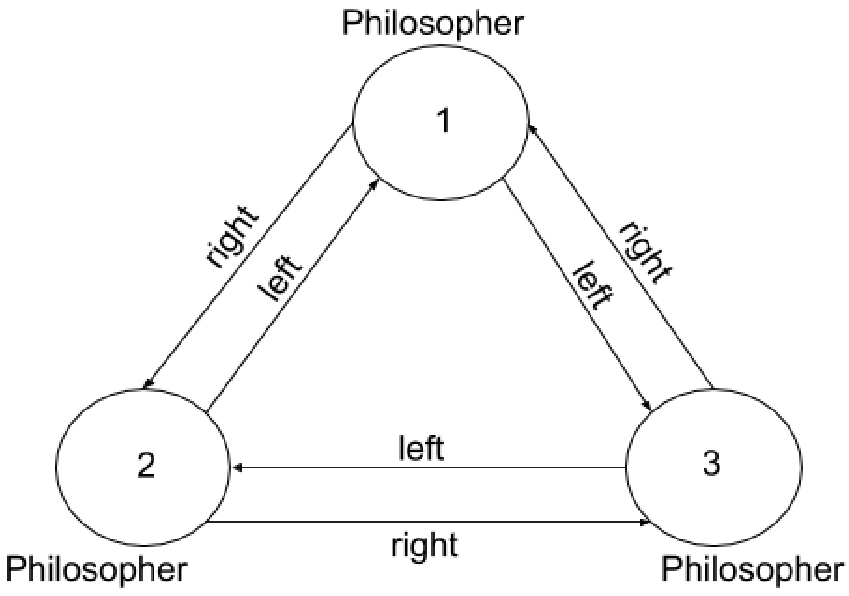
- $Philosopher_k[hashi_i, hashi_j] \overset{def}{=} Send_k[i,j] + Receive_k[l,r]$, with $k \in \{i,j\}$, it means that a philosopher can only send or receive hashis at each reduction of the process and, by definition, the *Send* e *Receive* processes can be expressed as follows:

$$Send_k[i,j] \overset{def}{=} Send(\overline{right}, \overline{left})[hashi_i, hashi_j]$$
$$Receive_k[l,r] \overset{def}{=} Receive(right, left)\ [hashi_l, hashi_r].$$

- $Dining_n \overset{def}{=} Philo_1[1]\ |\ Philo_2[2]\ |\ \cdots\ |\ Philo_n[n]$, is the process that represents the dining Chinese philosopher's system running.

Limiting dinner to three philosophers, as requested in the problem, we have the situation represented by Figure 3.2:

Figure 3.2: Dining to three Chinese Philosophers Problem.



From the figure, it can observe that the communication between the philosophers is limited. Philosopher one, for example, can communicate on the right only with philosopher two and on the left only with philosopher three. Therefore, it is natural to use as a resource in the Dining process reduction, variables identified with indices "r" (right), and "l" (left).

Another observation that can be made is that after the first philosopher passes his hashi to the right or left, the following situation will happen during the reductions: one of the philosophers will have no hashi, another will have only one hashi, and the third will two. Thus, only one philosopher can dine with each movement, while others will be in a state of thought.

Using the definitions above for a dinner with three Chinese philosophers we will evaluate the reduction of the system defined by the process $Dining_3 = Philo_1[1] \mid Philo_2[2] \mid Philo_3[3]$, passing the values $hashi_1$, $hahsi_2$ and $hashi_3$. The reductions are a result of applying the equations defined in the previous sections and expressed in Table 3.3. To organize, and better understand the reductions, the behavior of environment and memory have been placed separately in Table 3.4. Each n^o in Table 3.3 corresponds to the same n^o in Table 3.4.

Table 3.3: The behaviour of the system when one philosopher chose to send your hashi to right philosopher. Reduction by $Dining_n$, with $n = 3$.

n^o	Process
1.	$trans(Dining_3) = trans(Philo_1[1] \mid Philo_2[2] \mid Philo_3[3])$
2.	$= trans(Philo_1[1,0] \mid Philo_2[2,0]) \mid trans(Philo_3[3,0])$
3.	$= trans(Philo_1[1,0]) \mid trans(Philo_2[2,0]) \mid trans(Philo_3[3,0])$
4.	$= sum(Send_1[1,0])\ (\bar{r}\,1) \mid sum(Receive_2[2,r])\ (r\,1) \mid trans(Philo_3[3,0])$
5.	$= trans(actpref(Send_1[1,0]\ (\bar{r}\,1))) \mid trans(actpref(Receive_2[2,r]\ (r\,1)) \mid trans(Philo_3[3,0])$
6.	$= trans(Philo_1[0,0]) \mid trans(\{{}^{hashi_1}/_{hashi_r}\}Philo_2[2,1]) \mid trans(Philo_3[3,0])$
7.	$= trans(Philo_1[\,]) \mid trans(Philo_2[2,1]) \mid trans(Philo_3[3])$
8.	$= sum(Receive_1[l,r])\ (l\,1) \mid sum(Send_2[2,1])\ (\bar{r}\,2)\ (\bar{l}\,1) \mid sum(Receive_3[3,r])\ (r\,2))$
9.	$= trans(actpref(Receive_1[l,r]\ (l\,1))) \mid trans(actpref(Send_2[2,1]\ (\bar{r}\,2)\ (\bar{l}\,1))) \mid trans(actpref(Receive_3[3,r]\ (r\,2)))$
10.	$= trans(\{{}^{hashi_1}/_{hashi_l}\}Philo_1[1,r]) \mid trans(Philo_2[0,0]) \mid trans(\{{}^{hashi_2}/_{hashi_r}\}Philo_3[3,2])$
11.	$= trans(Philo_1[1]) \mid trans(Philo_2[\,]) \mid trans(Philo_3[3,2])$
12.	$= sum(Receive_1[1,r])\ (r\,3)) \mid sum(Receive_2[l,r])\ (l\,2) \mid sum(Send_3[3,2])\ (\bar{r}\,3)\ (\bar{l}\,2)$
13.	$= trans(actpref(Receive_1[1,r]\ (r\,3))) \mid trans(actpref(Receive_2[l,r]\ (l\,2))) \mid trans(actpref(Send_3[3,2]\ (\bar{r}\,3)\ (\bar{l}\,2)))$
14.	$= trans(\{{}^{hashi_3}/_{hashi_r}\}Philo_1[1,3]) \mid trans(\{{}^{hashi_2}/_{hashi_l}\}Philo_2[2,r]) \mid trans(Philo_3[0,0])$
15.	$= trans(Philo_1[1,3]) \mid trans(Philo_2[2]) \mid trans(Philo_3[\,])$
16.	$= sum(Send_1[1,3])\ (\bar{r}\,1)\ (\bar{l}\,3) \mid sum(Receive_2[2,r])\ (r\,1)) \mid sum(Receive_3[l,r])\ (l\,3)$
17.	$= trans(actpref(Send_1[1,3]\ (\bar{r}\,1)\ (\bar{l}\,3))) \mid trans(actpref(Receive_2[2,r]\ (r\,1))) \mid trans(actpref(Receive_3[l,r]\ (l\,3))$
18.	$= trans(Philo_1[0,0]) \mid trans(\{{}^{hashi_1}/_{hashi_r}\}Philo_2[2,1] \mid trans(\{{}^{hashi_3}/_{hashi_l}\}Philo_3[3,r])$
19.	Repeat equation n^o 7.: $= trans(Philo_1[\,]) \mid trans(Philo_2[2,1]) \mid trans(Philo_3[3])$

By applying the equations in $Dining_3$, we can see from both Table 3.3 and Table 3.4 that at a given moment the result begins to repeat, equation n^o 19 is equivalent to equation n^o 7. The next memory update will be equivalent to Equation 9. from Table 3.4. It means that the process is recursive, and the leaves of the synchronized tree representing this system begin to repeat.

Table 3.4: Behaviour of the memory and environment in the Table 3.3 reductions.

n^o	Environment	Memory
1.	ϵ	m
2.	ϵ	m
3.	ϵ	m
4.	ϵ	m
5.	$overlay(\epsilon, Link(r)[hashi_r]$	$update(m, L(r)[\,], R(r)[hashi_1])$
6.	$\epsilon[L(r)[hashi_r]]$	$m[^{hashi_1}/_{hashi_r}]$
7.	ϵ'	m'
8.	ϵ'	m'
9.	$overlay(overlay(\epsilon', L(r)[hashi_r]), L(l)[hashi_l])$	$update(update(m', L(r)[\,], R(r)[hashi_2]), L(l)[\,], R(l)[hahsi_1])$
10.	$\epsilon'[L(r)[hashi_r], L(l)[hashi_l]]$	$m'[^{hashi_2}/_{hashi_r}, {}^{hashi_1}/_{hashi_l}]$
11.	ϵ''	m''
12.	ϵ''	m''
13.	$overlay(overlay(\epsilon'', L(r)[hashi_r]), L(l)[hashi_l])$	$update(update(m'', L(r)[\,], R(r)[hashi_3]), L(l)[\,], R(l)[hahsi_2])$
14.	$\epsilon''[L(r)[hashi_r], L(l)[hashi_l]]$	$m''[^{hashi_3}/_{hashi_r}, {}^{hashi_2}/_{hashi_l}]$
15.	ϵ'''	m'''
16.	ϵ'''	m'''
17.	$overlay(overlay(\epsilon''', L(r)[hashi_r]), L(l)[hashi_l])$	$update(update(m''', L(r)[\,], R(r)[hashi_1]), L(l)[\,], R(l)[hahsi_3])$
18.	$\epsilon'''[L(r)[hashi_r], L(l)[hashi_l]]$	$m'''[^{hashi_1}/_{hashi_r}, {}^{hashi_3}/_{hashi_l}]$
19.	ϵ'^{v}	m'^{v}

Exercise 1. *Show that if $Dining_3$ starts with a philosopher sending his hashi to the philosopher on his left, the result of reductions will be equivalent if the philosopher sends his hashi to the right philosopher.*

Proof *By hypothesis, we can assume that if the right philosopher changes places with the left philosopher, the result will be the same, that is, the synchronisms with the right and left philosophers are equivalent.*

Applying the function trans in the process $Dining_3$ one have two possible results:

$$\begin{aligned} trans(Dining_3) &= trans(Philo_1 \mid Philo_2 \mid Philo_3) \\ &= sum(\boldsymbol{\tau} \,.\, (Philo_1 \mid Philo_2) \mid Philo_3)\ \boldsymbol{\tau} \\ &+ sum(\boldsymbol{\tau} \,\boldsymbol{.}\, (Philo_1 \mid Philo_3) \mid Philo_2)\ \boldsymbol{\tau} \end{aligned}$$

i.e.:

1^0. *if* $Philo_1$ *send to right* : $\tau\ .\ (Philo_1 \mid Philo_2) \mid Philo_3 \xrightarrow{\tau} \tau\ .\ (Philo_1 \mid Philo_2 \mid Philo_3)$

2^0. *if* $Philo_1$ *send to left* : $\tau\ .\ (Philo_1 \mid Philo_3) \mid Philo_2 \xrightarrow{\tau} \tau\ .\ (Philo_1 \mid Philo_3 \mid Philo_2)$

For the Structural Congruence π*-Calculus by Milner (1999):*

$$Philo_3 \mid Philo_2 \equiv Philo_2 \mid Philo_3$$

Then,

$$\tau\ .\ (Philo_1 \mid Philo_3 \mid Philo_2) \equiv \tau\ .\ (Philo_1 \mid Philo_2 \mid Philo_3)$$

For the Reaction Rule STRUCT by Milner (1999):

$$\tau\ .\ (Philo_1 \mid Philo_2) \mid Philo_3 \equiv \tau\ .\ (Philo_1 \mid Philo_3) \mid Philo_2$$

Therefore, if $Philo_1$ *sends his hashi to your left or right, system behavior is equivalent.*

Exercise 2. *Show that next memory update will be equivalent to Equation 9. from Table 3.4.*

Proof *Two memories are equivalent if your locations sets are equally allocated (MILNER, 1984). According Zermelo (1908), the Extension Axiom: "both sets A and B are equals if and only if A* $\subseteq$ *B and B* $\subseteq$ *A". Let* $m^v, m' \subseteq Memory$ *be a subset of Memory, then* $m^v = m''$ *if in* m^v *contains all substitutions in* m'' *and vice versa.*

Considering the structure of the Dining process, the memory m uses only two memory positions initially undefined: $L(r)[hashi_r]$*, associated to the right link and* $L(l)[hashi_l]$*, associated to left link, which will be updated during the reduction of this process. There is that:*

$$m'' = update(update(m';\ L(r)[hashi_r];\ R(r)[hashi_2]);\ L(left)[hashi_l];\ R(l)[hahsi_1])$$

Then, the memory m'' *set is* $m'' = \{[{}^{hashi_2}/_{hashi_r}], [{}^{hashi_1}/_{hashi_l}]\}$*. The next memory update will be as follows:*

$$udate(update(m'^v, L(r)[hashi_r], R(r)[hashi_2]), L(l)[hashi_l], R(l)[hashi_1]) = m'^v[{}^{hashi_2}/_{hashi_r}, {}^{hashi_1}/_{hashi_l}]$$

Therefore the two spaces allocated in the m'^v *update are allocated in the same way as in* m''*, which implies that* $m^v = \{[{}^{hashi_2}/_{hashi_r}], [{}^{hashi_1}/_{hashi_l}]\} = m'' \Leftrightarrow m^v \equiv m''$*.*

4 TYPED π-CALCULUS

A type system has the property *type* capable of classifying expressions in a language. Using a set of rules determined by the instantiated type, one can get an approximate description of possible values that buildings can store or compute, or the messages they will carry.

The main objective of a type system in π-*Calculus* is to reduce errors by defining interfaces between different parts of a process, and then to verify that the synchronisms occur consistently. In this section, a new syntax on *Basic* π-*Calculus* will be introduced to describe *Typed* π-*Calculus*.

4.1 Defined Types

To define the types in π-*Calculus*, one must first consider a set formed by all the basic types already known and instantiated. To maintain the coherence of this work, we will use the types already instantiated in MAUDE, according to Clavel et al. (2003):

$$Base = \{Nat, Int, Bool, String, ...\} \tag{4.1}$$

The π-*Calculus* define two types for labels. The first is the value type, called *VType*, and defined as follows:

$$VType \;::=\; Base \tag{4.2}$$

The value type is formed by all known types and has as a characteristic, labels capable of being sent or received by a communication channel (link). As in π-*Calculus*, there is the mobility of communication channels, and links can also assume this capacity. The function *typeval* can tell what the specified type of Base is:

$$typeval : VType \longrightarrow Base \tag{4.3}$$

For all $t \in VTtpe$ and $b \in Base$, the equations as follows;

$$typeval[\![\, t \,]\!] = b \tag{4.4}$$

The second is the link type, called *LType*:

$$LType \ ::= \ \mathbf{chan} VType \mid \mathbf{chan}\, LType, \tag{4.5}$$

formed by all types of values that assume the role of communication channels between processes within a type environment (defined below), capable of carrying values of a given value type. The function *typechan* can identify the Base type that a typo link is capable of carrying:

$$typechan : LType \longrightarrow Base \tag{4.6}$$

Let $l \in LType$ e $b \in VType$ be types, such that $l = \mathbf{chan}\ b$, the equations is as follows:

$$typechan[\![\ \mathbf{chan}\ b\ ]\!] = typeval(b) \tag{4.7}$$

$$typechan[\![\ \mathbf{chan}\ l\ ]\!] = typechan(l) \tag{4.8}$$

One can now define *Type* as the generalized composition of *VType* and *LType*. The *Type* set is necessary because of the π-*Calculus* mobility that allows values and links to be transported. Therefore, there are situations where a link is capable of carrying another link.

$$Type \ ::= \ VType \mid LType \tag{4.9}$$

Once defined *Type*, what matters is to know which type a label has and for which type a process is well defined. Therefore, we created a type environment that stores this information and which can be consulted so that the constructions are done correctly.

4.2 Type Environment

Let's define a type environment like a function from Label to Type. The behavior of a typed environment by Sangiorgi and Walker (2003) is as follows:

Definition 10. ***Type Environment*** *- An assignment of a type to a label is of the form*

$$a : Type,$$

where a is a Label, called the name of the assignment, and Type is a type, called the type of the assignment. A type environment is a finite set of assignment of types to names, where the *names on the assignments are all different. The Γ meta-variable will be used to represent a type environment.*

As stated earlier, what matters in a type environment is whether a label is of the type *VType* or *LType*. Therefore, it is possible to define the type environment as a function capable of saying which label assignment in this environment:

$$\Gamma : Label \longrightarrow Type \tag{4.10}$$

Let $x \in Label$, so:

$$\Gamma(Link(x)) = LType \tag{4.11}$$

$$\Gamma(name(x)) = VType \tag{4.12}$$

In Typed π-*Calculus*, it is assumed that each label can have a type. An *LType* label is able to manipulate typed values in a predefined environment. For example, if *b* is a type *VType* and x : **chan** b, then x is a link that can send or receive values of type *b*. A type environment can be syntactically defined as follows:

$$\Gamma \ ::= \ \mathbf{empty}\boldsymbol{\gamma} \ \big| \ \Gamma[Label : LType] \ \big| \ \Gamma[Label : VType] \ \big| \ \Gamma[Process] \tag{4.13}$$

Definition 11. *(Type Environment) Let $\gamma \in \Gamma$, $x, y \in Label$, $l \in LType$, $b \in VType$, $t \in Type$ and $P \in Process$ be variables, the informal semantic as follows:*

- ***empty**$\boldsymbol{\gamma}$ - the environment is empty;*

- *$\gamma[x : b]$ - in a type environment γ, x is a value of the value type b ($x : b$);*

- *$\gamma[y : l]$ - in a type environment γ, y is a channel of the link type l ($y : l$);*

- *$\gamma[P]$ - in a type environment γ, if a P process is well defined to type t, i.e., $\gamma[P : t]$ then for all x labels, such that $x \in bn(P)$, x is also of the type t in γ ($\gamma[x : t]$).*

4.2.1 Γ Type Environment: Semantic Functions and Equations

To reduce a process, it is necessary to "consult" the type environment. For this, one can have the function *show*, set to Γ able to show what type of labels are in a process, and the function *show-overlay* that updates an environment:

$$\mathit{show} \ : \Gamma \longrightarrow \mathit{Type} + \varnothing + \mathbf{wrong} \tag{4.14}$$

$$\mathit{show\text{-}overlay} : \Gamma \times \Gamma \longrightarrow \Gamma \tag{4.15}$$

Let $\gamma \in \Gamma$, $x, y \in \mathit{Label}$, $l \in \mathit{LType}$, $b \in \mathit{VType}$, $t \in \mathit{Type}$ and $P \in \mathit{Process}$ be the equations as follows:

$$\mathit{show\text{-}overlay}(\gamma, x : t) = \mathit{show}(\gamma'[x : t]) \tag{4.16}$$

$$\mathit{show}[\![\ \mathbf{empty}\boldsymbol{\gamma}\ ]\!] = \varnothing \tag{4.17}$$

$$\mathit{show}[\![\ \gamma[x : l]\ ]\!] = \Gamma(\mathit{Link}(x)) \tag{4.18}$$

$$\mathit{show}[\![\ \gamma[y : b]\ ]\!] = \Gamma(\mathit{name}(y)) \tag{4.19}$$

$$\mathit{show}[\![\ \gamma[P]\ ]\!] = \begin{cases} \mathit{if}\ \exists\ x \in \mathit{bn}(P)\ \mid\ x : t \\ \mathit{then}\ \Gamma(x) \\ \mathit{else}\ \mathbf{empty}\boldsymbol{\gamma} \end{cases} \tag{4.20}$$

$$\mathit{show}[\![\ \gamma[\mathbf{nill}]\ ]\!] = \mathbf{empty}\boldsymbol{\gamma} \tag{4.21}$$

$$\mathit{show}[\![\ \gamma[\bar{x}\, y\ \textbf{.}\ P]\ ]\!] = \begin{cases} \mathit{if}\ (\bar{x} : \mathbf{chan}\ t) \wedge (y : t \wedge \gamma[P]) \\ \mathit{then}\ \mathit{show}(\gamma[P]) \\ \mathit{else}\ \mathbf{wrong} \end{cases} \tag{4.22}$$

$$\mathit{show}[\![\ \gamma[x\, z\ \textbf{.}\ P]\ ]\!] = \begin{cases} \mathit{if}\ (x : \mathbf{chan}\ t) \wedge (\gamma[z : t, P]) \\ \mathit{then}\ \mathit{show}(\gamma[P]) \\ \mathit{else}\ \mathbf{wrong} \end{cases} \tag{4.23}$$

$$\mathit{show}[\![\ \gamma, [\boldsymbol{\tau}\ \textbf{.}\ P]\ ]\!] = \mathit{show}(\gamma[P]) \tag{4.24}$$

$$\mathit{show}[\![\ \gamma[[x = y]P]\ ]\!] = \begin{cases} \mathit{if}\ (x : \mathbf{chan}\ t) \wedge (y : \mathbf{chan}\ t) \wedge (\gamma[P]) \\ \mathit{then}\ \mathit{show}(\gamma[P]) \\ \mathit{else}\ \mathbf{wrong} \end{cases} \tag{4.25}$$

$$show[\![\ \gamma[P_1 + P_2]\ ]\!] = \begin{cases} \textit{if}\ (\gamma[P_1]) \wedge (\gamma[P_2]) \\ \textit{then}\ show(\gamma[P_1]) \vee show(\gamma[P_2]) \\ \textit{else}\ \textbf{wrong} \end{cases} \tag{4.26}$$

$$show[\![\ \gamma[P_1 \mid P_2]\ ]\!] = \begin{cases} \textit{if}\ \gamma[P_1] \wedge \gamma[P_2]) \\ \textit{then}\ show\text{-}overlay(\gamma[P_1], [P_2]) \\ \textit{else}\ \textbf{wrong} \end{cases} \tag{4.27}$$

$$show[\![\ \gamma[\textbf{new}\ x(P)]\ ]\!] = \begin{cases} \textit{if}\ (\gamma[x : l]) \wedge (\gamma[P]) \\ \textit{then}\ show\text{-}overlay(\gamma[x : l, P]) \\ \textit{else}\ \textbf{wrong} \end{cases} \tag{4.28}$$

$$show[\![\gamma[\textbf{Replicate}\ P]]\!] = show(\gamma[P]) \tag{4.29}$$

In expression 4.28, x is a bounded label restricted to the P process and is, therefore, typed. The restriction function creates a new communication channel used for synchronism and mobility in π-*Calculus*, so x is of type *LType* (type link) by definition.

Now it is possible to work with a type environment to the reduction of the processes. Through typed domains, one can define a values passing for the process transition. This will be presented in the next chapter.

5 VALUES PASSING

Since type labels are registered in a type environment, it is possible to define some domains according to the label's behavior in a process. Therefore, one can ensure the right value passing or the system is wrong. Follows the domains:

$$TLink = \{x \mid x \in Label \ \wedge \Gamma(x) \in LType\} \tag{5.1}$$

$$TMessage = \{y \mid y \in Message \wedge \forall y_n \in y, n \in \mathbb{N},\ \Gamma(y_n) \in Type\} \tag{5.2}$$

$$TIdentifier = \{i \mid i \in Identifier \wedge \Gamma(i) \in Type\} \tag{5.3}$$

Theorem 1. *TIdentifier* $\subseteq$ *TMessage.*

Proof *By definition, for all* $y \in$ *Message, y can be a value, a variable or a link. Let Identifier* $\subseteq$ *Message be a subset of variables, for all* $i \in$ *Identifier, then* $i \in$ *Message. So if* $i \in$ *Identifier is typed, it will be typed in Message. Then for all* $i \in$ *TIdentifier one has to* $i \in$ *TMessage* $\Rightarrow$ *TIdentifier* $\subseteq$ *TMessage.* $\square$

5.1 Values Passing During Parallel Composition

The π-*Calculus*, like CCS, assumes that Values Passing interactions take place between two agents. The difference over both is the capability of realizing these interactions as many times as necessary in π-*Calculus*. Taking into account the π syntax on expression 2.28, one can to reset any expressions considering the semantic domains as follows: Let $x \in$ *TLink*, $y \in$ *TMessage* and $z \in$ *TIdentifier* be then the follow prefix has the respective semantics meaning:

- output prefix: $\bar{x}\, y$ - x is the channel of communication. Therefore, $\Gamma(x) \in$ *LType* and x can send y of the type *VType* or *LType*. But this is only possible if the y value type is the same that x can transfer. For example, let $P = \bar{x}\ y\ .\ Q$ be a process defined to the type environment $\gamma[P : Nat]$. It means that the P process is well defined for type Nat and the transition of this process is only possible if $y : Nat$ is any value of the *Nat* type:

  if $(y : Nat) \wedge (x : \textbf{chan}\ Nat) \wedge (\gamma[Q])$

  then $sum(P\ (\bar{x}\, y))\ \epsilon\ m = trans(actpref(\bar{x}\, y\ .\ Q\ (\bar{x}\, y))\ \epsilon\ update(m;\ L(x)[\,];\ R(x)[y])$

  else **wrong**

Now suppose that let $y : l$ be a link type, with $l \in LType$, and make $\gamma[P]$ be a type environment, with P as in the previous example. Some observations must be considered for that transition to occur correctly. In this case,

$$\begin{aligned}&\mathit{if}\ (y : l) \wedge (x : \textbf{chan}\ l) \wedge (\gamma[Q])\\ &\mathit{then}\ \mathit{sum}(P\ (\bar{x}\ (y)))\ \epsilon\ m = \mathit{trans}(\mathit{actpref}(\bar{x}\, y\ .\ Q\ (\bar{x}\ (y)))\ \epsilon\ \mathit{update}(m;\ L(x)[\,];\ R(x)[y])\\ &\mathit{else}\ \textbf{wrong}\end{aligned}$$

and at this moment, it is not necessary to know which type y is capable to transfer. Probably its validity will be verified in a possible synchronism.

- input prefix: $x\ z$ - x is the channel of communication. Therefore, it is of the LType type and can receive a value or link of the Type type for z variable. But this is only possible if both the received value or link and z have the same type. For example, let $P = x\ z\ .\ Q$ be a process defined to the type environment $\gamma[z : Nat, P]$. It means that the Q process is well defined to Nat type and the transition of this process is only possible if x is a link with the capability to receive a variable of Nat type, the equations[1] as follows:

$$\begin{aligned}&\mathit{if}\ (x : \textbf{chan}\ Nat) \wedge (\gamma[z : Nat, Q])\\ &\mathit{then}\ \mathit{sum}(x\ z\ .\ Q\ (x\ 2))\ \epsilon\ m\ \ \overset{+}{=}\ \mathit{trans}(\{{}^{2}\!/\!{}_{z}\}Q')\ \epsilon[L(x)[z]]\ m[{}^{2}\!/\!{}_{z}]\\ &\mathit{else}\ \textbf{wrong}\end{aligned}$$

$\{{}^{2}\!/\!{}_{z}\}Q'$ means the process result of replacing all unbound occurrences of z in Q by 2.

As in the previous item, the passing of links must meet some criteria. In the example above, the transition occurs only if $y, z : l$ is of type link, with $l \in LType$. Also, all the linked occurrences of z in Q must be of the link type, that is Q must be well defined for the environment of type $\gamma[z : l]$. The transition as follows,

$$\begin{aligned}&\mathit{if}\ x : \textbf{chan}\ l \wedge \gamma[z : l, Q]\\ &\mathit{then}\ \mathit{sum}(x\ z\ .\ Q\ (x\ y))\ \epsilon\ m\ \ \overset{+}{=}\ \mathit{trans}(\{{}^{y}\!/\!{}_{z}\}Q')\ \epsilon[L(x)[z]]\ m[{}^{y}\!/\!{}_{z}]\\ &\mathit{else}\ \textbf{wrong}\end{aligned}$$

[1]($\overset{+}{=}$) The transitive and reflexive closure. It indicates the occurrence of more than one rewrite.

What guarantees π-*Calculus* mobility is the passage of communication channels (links), is contrast to CCS. We will see a complete example below.

- synchronise prefix: τ - when a synchronism occurs on a type environment, it means that the occurred substitution within a process was successful, and the process is well defined in the same type environment where the synchronism occurred. For example, let $P = x\,z \,.\, Q$ be a process defined to the type environment $\gamma[z : Int, P]$ and $R = \bar{x}\,y \,.\, S$ a process defined to the type environment $\gamma[R]$. It means that the P and R processes are defined for Int type and the transition of the $P \mid R$ composition is only possible if $y : Int$.

  For example, if one has $y = -1$ in the memory m, then:

  if $y : Int \wedge \gamma[z : Int, Q, S]$
  then $trans(x\,z \,.\, Q \mid \bar{x}\,y \,.\, S)\ \epsilon\ m[{}^{-1}\!/\!_{y}] \ \overset{+}{=}\ trans(\{{}^{-1}\!/\!_{z}\}Q' \mid S)\ \epsilon[L(x)[z]]\ m'[{}^{-1}\!/\!_{z}]$
  else **wrong**

  it means that the value -1 was assigned to z and the result this composition $\{{}^{-1}\!/\!_{z}\}Q' \mid S$ is the substitution of all occurrences of the variable z on Q by -1. The type environment update to $\gamma[Q' \mid S]$.

In the case of links, attention is needed with the restriction function (**new**). In synchronism, the link is restricted to the synchronized processes until the values pass through. If it is the link itself that is being moved, the restriction can end, and new restrictions can happen. This depends on the location of the link during the process transitions.

For example, let $P = x\,z \,.\, Q$ be a process defined to the type environment $\gamma[z : l, P]$ and $R =$ **new** $y(\bar{x}\,y \,.\, S)$ be a process defined to the type environment $\gamma[R]$. It means that the P and R processes are defined for $l \in LType$ link type and the transition of the $P \mid R$ composition is only possible if $y : l$. So,

if $y : l \wedge \gamma[z : l, \textbf{new}\ y(Q), S]$
then $trans(\textbf{new}\ y(\bar{x}\,y \,.\, S) \mid x\,z \,.\, Q)\ \epsilon\ m \ \overset{+}{=}\ trans(S \mid \textbf{new}\ y(\{{}^{y}\!/\!_{z}\}Q')\ \epsilon'\ m'[{}^{y}\!/\!_{z}]$
else **wrong**

where the y constraint has been transferred. In practice, communication between two processes is lost and the same communication starts in two other processes. Through replication, this transfer of communication channels can be recursive. But the proof of this is in the interest of the reader, as it can be easily concluded with the application of the theory and functions seen so far. Hence, you can define a set of expressions formed by:

$$\mathit{Expression} \ ::= \ \bar{x}\,\mathit{Exp}\,.\,\mathit{Process} \ \big|\ x\,\mathit{Var}\,.\,\mathit{Process} \ \big|\ \tau\,.\,\mathit{Process} \ \big|\ [\mathit{Exp} = \mathit{Exp}]\mathit{Process}, \quad (5.4)$$

where *Exp* are expressions of values that admit the reduction properties for the type instantiated and defined in the type environment or links, i.e., it can be evaluated, resulting in a value of one type. Likewise, *Var* are typed variables. Therefore, the passing of values only occurs when the reduction of the expressions arrives in the canonical form of the term with a defined type.

Semantics admit that the constructions are written correctly, so the result of the expressions will always have a type defined in a type environment. π-*Calculus* allows two forms of communication.

- Value-passing: through synchronise.
- Value-matching: ($[\mathit{Exp}_1 = \mathit{Exp}_2]\mathit{Process}$) one output expression Exp_1 matches with another Exp_2. Both expressions must return the same value and be of the same type, otherwise interaction is not possible.

5.2 Programming Language Constructs

According Fidge (1994), it is possible concluding some important constructions with *Expression*, that have to guarantee the functioning of the system. To $\bar{x}\ \mathit{Exp}\ .\ P$, with $P \in \mathit{Process}$, so:

- Conditional statements:

  $$\mathit{if\ Exp\ then\ P'\ else\ P''},$$

  where *Exp* is a boolean-valued expression or a restriction.

It means that the reduction of *Exp* arrives in the canonical form of the term, or the link is restricted. The scope of P is well defined in a type environment for this term, or the process is ill formed, and the error does not allow the process to run.

- Iteration

  $$\textit{while Exp do } P,$$

  where *Exp* is a boolean-valued expression or a restriction and is defined to a parameterised behaviour definitions.

  To x *Var* **.** P, with $P \in \mathit{Process}$, tem-se:

- Variable declaration:

  $$\textit{let Var} ::= \textit{Exp in } P ,$$

  where *Exp* is a boolean-valued expression or a restriction.

It means that *Var* can receive declared values in local memory, and if there is type compatibility, all linked instances of *Var* in *P* are replaced by the declared values.

5.3 Values Passing Examples

Looking at some examples of values passing during the reduction of processes.

Example 4. *(Expressions Reduction) Let* $P = x\,a \,\textbf{.}\, y\,b \,\textbf{.}\, \bar{z}\,(a - b + 1) \,\textbf{.}\, \textbf{\textit{nill}}$ *and* $Q = z\,c \,\textbf{.}\, S$ *be a process such that* $\gamma[P, Q : Int]$ *on type environment* γ*. Follows the reduction of the* $P \mid Q$ *in the environment* $\epsilon[L(x, y)[a, b]]$ *and the memory* $m[R(x)[-4], R(y)[5]]$*, where let* $a ::= -4$ *and* $b ::= 5$ *in P:*
In this case, the canonical form of the expression $Exp = a - b + 1$ *is equal to* -8 *of type Int. And the result* $\{^{-8}/_{c}\}S'$ *means that all linked occurrences of c in S have been replaced by* -8*. The rules for reducing the expression Exp derive from instantiating its type.*

By definition, a *Exp* can also be a link. In that case, some necessary conditions must be observed.

Example 5. *(mobility) Proceeding to mobility, let S,* $R = \bar{y}\,z \,\textbf{.}\, R'$*,* $P = y\,a \,\textbf{.}\, \bar{x}\,y \,\textbf{.}\, P'$ *and* $Q = x\,b \,\textbf{.}\, Q'$ *be process. To find a new direct connection between Q and R, and P and S, consider the composition* $P \mid Q$*.*

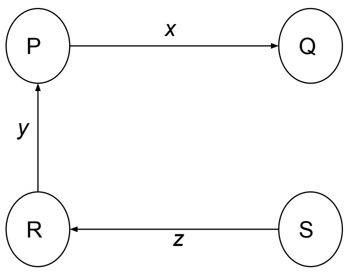
First one can $x, y, z \in LType$*. Second the link z is restricted to R and S, the link y is restricted to P and R, and the link x is restrict to P and Q. Using graph diagrams, one*

Table 5.1: Reduction process of the $P \mid Q$ with environment behaviour and memory state.

n^o	Process	Environment	Memory
1	$trans(P \mid Q) = trans(x\,a\,.\,y\,b\,.\,\bar{z}\,(a-b+1)\,.\,\mathbf{nill} \mid Q)$	$\epsilon[L(x,y)[a,b]]$	$m[{}^{-4}/_{a}, {}^{5}/_{b}]$
2	$= trans(x\,a\,.\,y\,b\,.\,\bar{z}\,(a-b+1)\,.\,Q) \mid trans(Q)$	$\epsilon[L(x,y)[a,b]]$	$m[{}^{-4}/_{a}, {}^{5}/_{b}]$
3	$= sum(x\,a\,.\,y\,b\,.\,\bar{z}\,(a-b+1)\,.\,\mathbf{nill})\ (x\ -4) \mid trans(Q)$	$\epsilon[L(x,y)[a,b]]$	$m[{}^{-4}/_{a}, {}^{5}/_{b}]$
4	$= trans(actpref(x\,a\,.\,y\,b\,.\,\bar{z}\,(a-b+1)\,.\,\mathbf{nill})\ (x\ -4)) \mid trans(Q)$	$\epsilon[L(x,y)[a,b]]$	$m[{}^{-4}/_{a}, {}^{5}/_{b}]$
5	$= trans(\{{}^{-4}/_{a}\}y\,b\,.\,\bar{z}\,(-4-b+1)\,.\,\mathbf{nill}) \mid trans(Q)$	$\epsilon[L(x,y)[a,b]]$	$m[{}^{-4}/_{a}, {}^{5}/_{b}]$
6	$= sum(y\,b\,.\,\bar{z}\,(-3-b)\,.\,\mathbf{nill})\ (y\ 5) \mid trans(Q)$	$\epsilon[L(x,y)[a,b]]$	$m[{}^{-4}/_{a}, {}^{5}/_{b}]$
7	$= trans(actpref(y\,b\,.\,\bar{z}\,(-3-b)\,.\,\mathbf{nill})\ (y\ 5)) \mid trans(Q)$	$\epsilon[L(x,y)[a,b]]$	$m[{}^{-4}/_{a}, {}^{5}/_{b}]$
8	$= trans(\{{}^{5}/_{b}\}\bar{z}\,(-3-5)\,.\,\mathbf{nill}) \mid trans(Q)$	$\epsilon[L(x,y)[a,b]]$	$m[{}^{-4}/_{a}, {}^{5}/_{b}]$
9	$= sum(\bar{z}\,(-8)\,.\,\mathbf{nill})\ (\bar{z}\ -8) \mid sum(z\,c\,.\,S)\ (z\ -8)$	$\epsilon'[L(z)[c]]$	$m'[R(z)[-8]]$
10	$=^{+} trans(\mathbf{nill}) \mid trans(\{{}^{-8}/_{c}\}S')$	$\epsilon'[L(z)[c]]$	$m'[{}^{-8}/_{c}]$
11	$= trans(\{{}^{-8}/_{c}\}S')$	$\epsilon'[L(z)[c]]$	$m'[{}^{-8}/_{c}]$

can represent this interaction with nodes as processes and arcs as restricted communication channels, the composition $P \mid Q \mid R \mid S$ has the following graph representation:

Figure 5.1: Graph representation of a continuous circuit from S to Q.



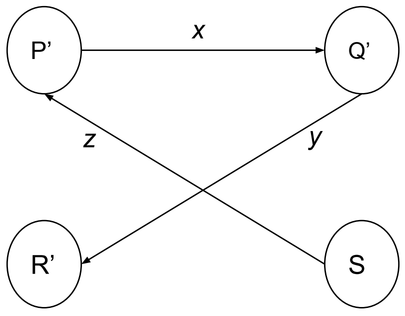
Looking the result of the process reductions:

Table 5.2: Reduction process of the $P \mid Q \mid R \mid S$.

n^o	Process	Environment	Memory
1	$trans(P \mid Q \mid R \mid S) = trans(\mathbf{new}\ x(Q \mid \mathbf{new}\ y(P \mid \mathbf{new}\ z(R \mid S))))$	ϵ	m
2	$=^{+} trans(\mathbf{new}\ x(Q \mid \mathbf{new}\ y(y\,a\,.\,\bar{x}\,y\,.\,P' \mid \mathbf{new}\ z(\bar{y}\,z\,.\,R' \mid S))))$	$\epsilon'[L(y)[a]]$	$m'[{}^{z}/_{a}]$
3	$= trans(\mathbf{new}\ x(x\,b\,.\,Q' \mid \mathbf{new}\ y(R' \mid \mathbf{new}\ z(\{{}^{z}/_{a}\}\bar{x}\,y\,.\,P' \mid S))))$	$\epsilon''[L(x)[b]]$	$m''[{}^{y}/_{b}]$
4	$=^{+} trans(\mathbf{new}\ x(\mathbf{new}\ y(\{{}^{y}/_{b}\}Q' \mid R') \mid \mathbf{new}\ z(\{{}^{z}/_{a}\}P' \mid S)))$	$\epsilon''[L(x)[b]]$	$m''[{}^{y}/_{b}]$

The result of the reduction of this composition represents a transition of channels, changing the restriction of the link y to Q' and R' and the restriction of the link z to P' and S, follows Fig. 5.2.

Figure 5.2: Graph representation of an alternating circuit from S to R'.



The two examples above show some possibilities of values passing simply and objectively. However, the π-*Calculus* is capable of more complex shapes according to the user's needs.

In general, the syntax and semantics presented so far provide the solution of more complex problems with the construction of complex algorithms. The mathematics behind this method proposes excellent reliability in the creation and development of critical systems that use competition, synchronism and mobility.

6 CONCLUSION

The π-*Calculus* method has been used in the industry, from the modeling of telecommunications protocols, to the modeling of object-oriented languages. It is because of its mobility capacity, parameter dependence, and dynamic reconfiguration. Despite covering a wide area of applicability, its popularity is small, since it has some flaws that hinder its use. This method has a strong mathematical base and many extensions, so the user needs comprehensive syntactic and semantic knowledge, to feel comfortable using it. Although it is complex, it is still advantageous, since it is less prone to errors, making it a good option for working with critical systems.

In an attempt to make it more accessible, the Pict (PIERCE; TURNER, 2000) programming language was developed, based on primitive terms, and then a more sophisticated core language, an asynchronous, choice-free fragment of the π-*Calculus* enriched with records and pattern matching. This programming language has a format that can be compiled by the same identifiers as the C programming language. However, still no formal language that can contribute to the creation of tools.

The denotational semantics developed in this work proposes a slightly more sophisticated version of the method that aims to facilitate its understanding and application. Hence, for each metavariable, a semantic function was built, containing as many definitions as there are alternatives to the metavariable. In addition, it provided the creation of the concept of memory and environment and verified its behavior in reducing processes. Through semantic domains, it was possible to show the passing values in the π-*Calculus*, since one of its peculiarities about the CCS is the passage of communication channels, maintaining and creating new labels.

6.1 Future Work

The contributions presented in this dissertation are a step forward for the development of a denotational semantic to the *Basic* π-*Calculus* and the *Typed* π-*Calculus*. Still, more extensions can be incorporated, enriching the possibility of developing a formal language. This study still has its limitations, which leads to future work opportunities.

Process Passing Once we have the passage of values and communication channels, it is possible to demonstrate the way of processes/agents. π-*Calculus* is capable of making this transition, and it also can be found in different literature, for example, Sangiorgi (1996).

Tool to π-*Calculus* Implementing a tool for π-*Calculus* is still a challenge. There are already some limited models, like MWB (VICTOR; MOLLER, 1994), an automated tool for manipulating and analyzing concurrent mobile systems described in the π-*Calculus* untyped.

Instantiate π-*Calculus* em Maude, in their study "Executing and verifying CCS in Maude", Verdejo, Martı-Oliet and Programación (2000), implemented CCS in Maude using **rl's** operators, commands that provide competition in the Maude language through rewrite rules. Through this implementation, the Maude Tool was able to run CCS. The idea is to implement π-*Calculus* in Maude using the denotational semantics developed in this dissertation.

REFERENCES

ABDALLAH, A. E. **Communicating Sequential Processes. The First 25 Years: Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004. Revised Invited Papers**. [S.l.]: Springer Science & Business Media, 2005.

BARENDREGT, H. P.; MANZONETTO, G.; PLASMEIJER, R. The imperative and functional programming paradigm. [Sl]: Elsevier, 2013.

BOLOGNESI, T.; BRINKSMA, E. Introduction to the iso specification language lotos. **Computer Networks and ISDN systems**, Elsevier, v. 14, n. 1, p. 25–59, 1987.

BUSTARD, D. W. **Concepts of Concurrent Programming**. [S.l.], 1990.

CLAVEL, M. et al. Maude 2.0 manual. **Available in http://maude. cs. uiuc. edu**, 2003.

COHEN, B.; HARWOOD, W. T.; JACKSON, M. I. **The specification of complex systems**. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 1986.

DIJKSTRA, E. W. Two starvation free solutions to a general exclusion problem. **Unpublished Tech. Note EWD**, v. 625, 1978.

FIDGE, C. A comparative introduction to csp, ccs and lotos. **Software Verification Research Centre, University of Queensland, Tech. Rep**, Citeseer, p. 93–24, 1994.

LAI, R.; JIRACHIEFPATTANA, A. **Communication protocol specification and verification**. [S.l.]: Springer Science & Business Media, 2012.

LOGRIPPO, L.; FACI, M.; HAJ-HUSSEIN, M. An introduction to lotos: learning by examples. **Computer Networks and ISDN systems**, Elsevier, v. 23, n. 5, p. 325–342, 1992.

MILNER, R. Lectures on a calculus for communicating systems. In: SPRINGER. **International Conference on Concurrency**. [S.l.], 1984. p. 197–220.

MILNER, R. **Communicating and mobile systems: the pi calculus**. [S.l.]: Cambridge university press, 1999.

MILNER, R.; PARROW, J.; WALKER, D. A calculus of mobile processes i, ii. **Information and computation**, Elsevier, v. 100, n. 1, p. 41–77, 1992.

NUNES, D. J. **Introdução a Abstração de Dados: Série Livros Didáticos Informática UFRGS**. [S.l.]: Bookman Editora, 2012.

PETERSON, J. L. Petri nets. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 9, n. 3, p. 223–252, 1977.

PIERCE, B. C.; TURNER, D. N. Pict: a programming language based on the pi-calculus. In: **Proof, language, and interaction**. [S.l.: s.n.], 2000. p. 455–494.

SANGIORGI, D. π-calculus, internal mobility, and agent-passing calculi. **Theoretical Computer Science**, Elsevier, v. 167, n. 1-2, p. 235–274, 1996.

SANGIORGI, D.; WALKER, D. **The pi-calculus: a Theory of Mobile Processes**. [S.l.]: Cambridge university press, 2003.

SLORMEGER, K.; KURTZ, B. **Syntax and Semantics of Programming Languages**. [S.l.]: New York: Addison Wesley, 1995.

TURNER, D. N. Polymorphic pi-calculus: theory and implementation. The University of Edinburgh, 1995.

VERDEJO, A.; MARTI-OLIET, N.; PROGRAMACIÓN, D. S. I. y. **Executing and verifying CCS in Maude**. [S.l.], 2000.

VICTOR, B.; MOLLER, F. The mobility workbench—a tool for the π-calculus. In: SPRINGER. **International Conference on Computer Aided Verification**. [S.l.], 1994. p. 428–440.

WATT, D. A. **Programming language syntax and semantics**. [S.l.]: Prentice Hall PTR, 1996.

ZERMELO, E. Untersuchungen über die grundlagen der mengenlehre. i. **Mathematische Annalen**, Springer, v. 65, n. 2, p. 261–281, 1908.

APPENDIX A — RESUMO ESTENDIDO

A verificação formal tem, cada vez mais, um papel importante no desenvolvimento de software pela indústria. O aprimoramento de linguagens formais e a criação de ferramentas que as comportam contribuem para este cenário, pois propiciam segurança no desempenho de sistemas, diminuem a necessidade de atualizações por conta de erros de execução e bugs, assim como, equilibram a balança do custo/benefício em relação ao tempo gasto com uma especificação formal.

Em outras palavras, o aprimoramento de linguagens ou métodos formais amplia sua aplicabilidade uma vez que abrange mais recursos. Portanto, o campo de possibilidades de escolha para o uso da melhor linguagem ou método se estende e o usuário pode aplicar um filtro que melhor se enquadra nas características e demandas de um sistema. Além disso, a formação de profissionais na área se torna mais acessível e as empresas podem investir em qualidade sem muitos importunos.

A criação de ferramentas também se torna essencial, pois, junto com o aprimoramento das linguagens e métodos formais, facilita a utilização da verificação formal de softwares por usuários leigos ou em formação. Uma vez que propicia testadores de escrita e provadores de teoremas, poupa tempo, garante o mínimo de erros e acelera o processo de comunicação entre linguagens formais e linguagens de programação.

Por essas e outras razões, a verificação formal é bastante indicada para a instanciação de sistemas críticos, uma vez que uso de linguagens formais no desenvolvimento desses sistemas assegura o funcionamento do sistema. Ao mesmo tempo, o custo/benefício do tempo usado na verificação formal de uma especificação de um sistema é menor quando se tem uma linguagem/método formal bastante rico por extensões e quando apresentam ferramenta que o suporte.

Atualmente, muitos métodos já estão bastante enquadrados nesta perspectiva. Mesmo classificados entre algébricos, concorrentes ou baseados em modelos, muitos já caminham entre essa classificação utilizando de extensões para se enquadrar em outros campos. Por exemplo, PN's é um método concorrente, porém possui características de métodos baseados em modelos e, por conta disso, possui muitas ferramentas sendo bastante popular entre os usuários de verificações formais. Outros exemplos são os métodos algébricos Maude e Lotos que usam de extensões para fazer concorrência.

Essa dissertação tem como propósito propiciar esta mesma perspectiva para o método formal π-*Cálculo* . O método formal π-*Cálculo* é muito usado em sistemas de transição rotulada e sistemas móveis, pois apresenta concorrência, mobilidade e criação de novos links. Porém, por mais abrangente que seja sua aplicabilidade, apresenta algumas limitações ao usuário como, por exemplo, não possui uma linguagem formal e nem ferramenta que o suporte. O π-*Cálculo* é uma extensão do CCS que apresenta semântica operacional, entretanto sem passagens de valores. Isso significa que o usuário precisa maior domínio de especificações nesse método formal para que consiga sucesso na sua utilização. Além de possuir bastante familiaridade com provas de teoremas, pois, ele se torna bastante "braçal" sem uma ferramenta.

Existe na literatura uma tentativa de linguagem de programação para o π-*Cálculo*, chamada Pict, baseada na semântica operacional do CCS, mas que ainda está em fase de teste. Para facilitar o processo de propiciar uma maior usabilidade do método formal π-*Cálculo*, é importante traçar uma "rota segura". Isso significa que a criação de uma linguagem formal sobre esse método e uma ferramenta que o suporte pode garantir que um sistema possa ser instanciado utilizando todos os recursos do método e suas extensões. Por isso, instanciar o π-*Cálculo* utilizando a semântica denotacional, baseada no mapeamento de frases sintáticas em domínios semânticos através de funções semânticas, oferece estrutura para a criação de uma linguagem formal e, consequentemente, a implementação de ferramentas, como no caso do LOTOS e do Maude.

Em síntese, esta dissertação consiste em proporcionar ao método formal π-*Cálculo* uma abordagem denotacional e como se dá a passagem de valores. Nesta abordagem o método foi divido em: Basic π-*Calculus* e Typed π-*Calculus*. A primeira contendo os conceitos básicos da construção e instanciação do método e a segunda, contendo uma extensão para o método baseada em tipos abstratos de dados. A passagem de valores foi feita sobre um ambiente de tipos, garantindo a tipagem correta de termos durante a transição de processos tipados.

A semântica denotacional desenvolvida neste trabalho propõe uma versão um pouco mais sofisticada do método que visa facilitar sua compreensão e aplicação. Consequentemente, para cada metavariável, uma função semântica foi construída, contendo tantas definições quantas forem as alternativas para a metavariável. Além disso, propiciou a criação do conceito de memória e ambiente, verificando seu comportamento na redução de processos. Por meio dos domínios semânticos, foi possível mostrar a passagem de valores no π-*Cálculo*, visto que uma de suas particularidades em relação ao CCS é a passagem de canais de comunicação, mantendo e criando novos rótulos.