Bachelor Thesis

Faculty of Science and Engineering

Computing Science

# Strong Normalization in Message-Passing Concurrency

*Author:*
Evrikli Xhelo

*Supervisors:*
Jorge Pérez
Gerard Renardel de Lavalette


July 2019

# STRONG NORMALIZATION IN MESSAGE-PASSING CONCURRENCY

EVRIKLI XHELO S3202267

---

## Contents

## 1. ABSTRACT

**This bachelor thesis is concerned with the termination property of message-passing concurrent systems. Type systems provide an approach to studying the behaviours of running processes and determining their resource usage and capability to terminate successfully. However, no standard rules can be defined so as to guarantee that a process will finish executing without having to execute it in the first place. We study two type systems that have been proposed for concurrent processes. More precisely, we have worked with one type system that is based upon intuitionistic linear logic from Caires, Pfenning and Toninho, and the core type system from Deng and Sangiorgi. Both papers propose their own approach to process and channel types, as well as typing rules, and both papers use formalisms based on $\pi$-calculus. We seek to determine how the two systems relate to one another and whether they share similarities in their respective approaches to typing terminating processes.**

## 2. Introduction

Normalization, also known as termination, remains one of the most important properties to study in the world of software. It is not possible to conclude whether every program is able to terminate successfully given certain inputs, a situation also known as the *halting problem* [Lan, 2019]. In this paper, we focus on message-passing concurrent processes: processes that run simultaneously while making possible the delivery and reception of other messages along different channels. A *process* is a program designed with a specific goal. Concurrent processes reflect important real-life situations as nowadays people are constantly sending and receiving multiple requests simultaneously, be it on our phones or work computers. Therefore, it is important to be able to understand the structures of these requests and whether they will be able to terminate successfully. Concurrent processes run simultaneously and in doing so, they are able to communicate with one another, should a common port between the two exist. We can think of a process as being a composition of several events; once an event is over, another one begins. There could also be events that happen recurrently, such as a server that is constantly waiting for requests from its clients. The driving force in processes are *channels* because they allow for interactions to happen. Channels are ports where messages get sent back and forth. They allow processes to evolve, that is, allow for one event to finish, so that the next one can begin. In concurrent processes, this notion is called synchronization.

How can we study the operations of processes and their evolution? We rely on the operational semantics of processes. Reducing a process is analogous to running a program on the computer. We calculate every step until we reach the final one, thus the step where the process terminates. Since it is not possible to know whether a process will terminate without rendering the reduction steps, we work around the impediment that the halting problem poses by making use of static techniques. That is, we develop means to prove the termination of certain processes without having to run them first. This is where type systems come in handy.

Essentially, a *type system* assigns *a type* to every channel on a process and relies on a set of well-defined rules for any possible operation. The type system of a programming language is used to prevent the occurrence of execution errors during the running of a program [Cardelli, 2004]. For instance, in C, every variable is defined by a type (e.g. int or char) and every operation is well-defined in terms of which types it will be applied to [B. MacQueen, 2012]; it is not possible to sum an integer and an array of characters. Therefore, in a type system, it is not possible to have anomalies: any viable variable (channel, port) and interaction in a process is clearly defined and

based on the set of well-defined typing rules. In this paper, we will consider type systems where a well-typed process ensures strong normalization. Therefore, proving that a process is well-typed in a type system using its well-defined rules, implies that the process is strongly normalizing.

Several type systems have been developed whose rules ensure that well-typed processes will always terminate. However, is it possible to type any possible (and terminating) process in these type systems? Are there processes that are well-typed in one type system, but fail to match the rules of another one? This bachelor thesis is about the research on two type systems for concurrent processes. More specifically, it is based on the study of two papers: "Towards Concurrent Type Theory" from Luis Cares, Frank Pfenning, Bernardo Toninho [Caires et al., 2012] and "Ensuring termination by typability" from Yuxin Deng and Davide Sangiorgi [Deng and Sangiorgi, 2006]. We seek to detect a subset of processes that are strongly normalizing by making use of the type systems.

We will firstly provide an introduction to $\pi$-calculus, the 'programming language' for representing concurrent processes and addressing issues related to concurrency [Deng and Sangiorgi, 2006]. Then, we will introduce the two type systems separately, along with their rules, main concepts and definitions. Afterwards, we will present three different processes that are seemingly well-typed in one type system, but not in the other. Finally, we conclude this paper with the insights and conclusions gained from our research.

## 3. $\pi$-Calculus

Process calculi are a class of formal systems that are used to mathematically express concurrent processes [Milner et al., 1992]. In this paper, we study one particular class, $\pi$-calculus. Other leading examples of process calculi are CSP and CCS. There are various notations to the syntax that $\pi$-calculus provides. However, in this section we use the same syntax that is used in one of the type systems that will be introduced further ahead.

In the $\pi$-calculus, every expression denotes a process [C. Pierce, 1995], named after upper-case letters of the alphabet, e.g. $P, Q$ or $R$. The basic computational step in $\pi$-calculus is the transfer of a communication link between two processes [Parrow, 2001]. Channels are responsible for creating this link. They are named after lower-case letters of the alphabet: $a, b, \ldots x, y, z$. Two processes can interact by exchanging messages on a (common) channel [C. Pierce, 1995]. A message could be an object, like an integer $43$ or string $\mathtt{hello}$, but at the same time, it could also be another channel. In a real-life example, let us suppose that in order to use a particular printer, a client needs certain information from the server. When the client makes a request to a server, the server receives the request and responds to the client. Once the client has received the necessary data from the server, it can finally proceed using the printer. We can represent the concurrent processes that occur here using $\pi$-calculus in the following way:

$$\overline{b}\langle a\rangle.S \mid b(c).\overline{c}\langle d\rangle.P^{1}$$

Channel $b$ serves as the communication link between the *server* side on the left, and the *client* side on the right. When the message across $b$ has been sent, the client uses it to transmit some data $d$ and then proceeds using the printer. After the interaction occurs, we have the following situation:

$$S \mid \overline{a}\langle d\rangle.P.$$

[1]Symbol $\mid$ is used to represent parallel composition, that is to showcase that the process on the left is running in parallel with the process on the right. We will describe the syntax in greater detail in §3.1.

The fact that a message can also be used as a communication link between processes, is one of the cornerstones of the calculus and an important point of differentiation with other process calculi [Parrow, 2001]. We will now introduce the syntax of the language, followed by the rules of structural congruence and operational semantics. The terms *channels* and *variables* are used interchangeably across the paper to represent both, messages and communication links.

3.1. **Syntax.**

| | | |
|---|---|---|
| $P, Q, R ::=$ | $\mathbf{0}$ | inert process |
| | $x(y).P$ | input prefix |
| | $\overline{x}\langle y\rangle.P$ | output prefix |
| | $P \mid Q$ | parallel composition |
| | $(\boldsymbol{\nu} x)P$ | restriction |
| | $!x(y).P$ | replication |

The simplest $\pi$-calculus expression is the *inert process* $\mathbf{0}$ [C. Pierce, 1995]. We use it to express a process that has no behavior, that is, a process that does not do anything.
An *input prefix*, $x(y).P$, denotes a process that inputs $y$ along $x$ and continues as $P$. In this process, $y$ is a bound variable. This means that its value in $P$ depends on and will be substituted for the variable that will be passed along $x$.
*Output prefix*, $\overline{x}\langle y\rangle.P$, denotes a process that outputs $y$ along $x$ and then resumes as $P$. In this situation, $y$ is a free variable, that is, a variable that does not have a bound occurrence. We can denote all the free variables in $P$ as $fn(P)$. The input and output channels are also called *subjects*, whereas the message that they receive and send respectively, are called *objects*.
*Parallel composition*, $P \mid Q$, denotes a process that is composed of two sub-processes running simultaneously. This process can exhibit every behavior of each sub-process, at any order whatsoever. When $P$ and $Q$ share channels in common, where one of them sends a message and the other receives it, $P \mid Q$ showcases an internal communication in which messages are exchanged between the two sub-processes.
*Restriction*, $(\boldsymbol{\nu} x)P$, represents the creation of a new channel in a process $P$. This channel is local to $P$ and cannot be used by other channels as a direct link to communicate with $P$ [Parrow, 2001], that is, its scope is process $P$. This process also binds $x$ in $P$. Thus, both restriction and input prefix, bind variables in a process. We can denote all the bound variables in $P$ as $bn(P)$. If there is also a free channel named $x$ in $P$, then we can easily rename the bound channel $x$ so that no message received through it will be mixed with those received on the free channel. In $\pi$-calculus, we call this alpha-conversion.
The *replication* process, $!x(y).P$, represents an infinite number of copies of itself. Each $P$ gets activated every time $y$ is received along $x$. Each copy runs simultaneously with the other ones that have been activated. The replication process can used to characterize a real-life server which receives input from a client, responds to that request and awaits new requests from other clients.

3.2. **Structural Congruence.** The notion of structural congruence is central to $\pi$-calculus as it describes an equivalence between processes. Two processes are structurally congruent if they are identical up to the structure [Milner et al., 1992]. We write $P \equiv Q$ to state that process $P$

is structurally congruent to process $Q$. We define structural congruence as the smallest relation between two processes that is based on the following rules:

– *Alpha-conversion*: $P \equiv Q$ if Q can be obtained from P by renaming the bound names in P.
– *Parallel composition*:
  - $P \mid Q \equiv Q \mid P$
  - $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
  - $P \mid \mathbf{0} \equiv P$

  The first structural rule for parallel composition states that rearranging the order of two parallel processes will not have any effects on the main process, i.e. parallel processes are commutative; the second rule states that parallel processes are associative. The third rule states that a process running simultaneously with the inert process is equivalent to that process running on its own.
– *Restriction*: $(\boldsymbol{\nu} x)P \mid Q \equiv (\boldsymbol{\nu} x)(P \mid Q)$
  This rule states that the scope of the restriction that binds $x$ to a process $P$, can be extended to bind $x$ to a process $Q$ that is running in parallel with $P$, as long as $x$ is not a free name in $Q$.

3.3. **Operational Semantics.** The operational semantics of $\pi$-calculus describes how a process evolves from one state to the other. In this paper, we consider reduction of a process under the communication and replication rule:

$$\overline{x}\langle y\rangle.P \mid x(z).Q \to P \mid Q\{y/z\} \qquad \text{communication}$$

$$(\boldsymbol{\nu} x)(!x(z).Q \mid (\boldsymbol{\nu} y)\overline{x}\langle y\rangle.P) \to (\boldsymbol{\nu} y)(Q\{y/z\} \mid (\boldsymbol{\nu} x)(!x(z).Q \mid P)) \qquad \text{replication}$$

Reduction under the condition of communication states that, if an output process $\overline{x}\langle y\rangle.P$ runs in parallel with an input process $x(z).Q$, then they will reduce to $P$ and $Q$ respectively, with $Q$ having every free instance of $z$ substituted for $y$. This happens because the processes have channel $x$ in common. The server-client example we introduced before is an application of this reduction rule. Another example in $\pi$-calculus would be:

$$\overline{x}\langle y\rangle.\mathbf{0} \mid x(b).\overline{b}\langle c\rangle.a(b) \longrightarrow \mathbf{0} \mid \overline{b}\langle c\rangle.a(b)\{y/b\} \equiv \overline{y}\langle c\rangle.a(b)$$

Note that the second $b$ is bound by the input channel $a$ and cannot be substituted by the reduction step. In this example, we write $\{y/b\}$ to state that every free instance of $b$ in $\overline{b}\langle c\rangle.a(b)$ is substituted for $y$. The result is then $\overline{y}\langle c\rangle.a(b)$. We will now present two type systems about concurrent processes. We will describe their rules and how we can type processes in each one of them.

Reduction under the condition of replication states that, if a process is a composition of two sub-processes that are running in parallel, where:

- one process offers the service of a replicated input in the form of a server, $!x(z)$, and then continues as $Q$ - this is the server process,
- the other process activates the server, $\overline{x}\langle y\rangle$, and then resumes as $P$ - this is the client process,

then the main process reduces to a composition of three sub-processes that are running in parallel:

- a copy of sub-process $Q$ where every instance of $z$ is substituted with $y$. $Q$ is the response of the server to the original client.
- a sub-process $P$. This is the process that client resumes as.
- a sub-process that offers the service of a replicated input, $!x(z).Q$. This sub-process showcases the resumption of the server to accepting other requests.

With the operational semantics, we conclude the introductory section about $\pi$-calculus. We discussed the main concepts that are relevant to this thesis. We will now present the two type systems that this research was concerned with, starting firstly with the type system introduced by Caires, Pfenning, and Tonihno, and then with the one introduced by Deng and Sangiorgi.

## 4. The CPT Type System

Caires, Pfenning, and Tonihno have provided their approach to how a typed version of $\pi$-calculus can be used as a type system for concurrency [Caires et al., 2012]. They have introduced a type system that is based on sequent style, intuitionistic linear logic. Throughout this paper, we will refer to this type system as CPT. CPT satisfies the properties of the Curry-Howard correspondence, which entails that any process that is well-typed in it, will terminate [Caires et al., 2012], that is, is strongly normalizing.

4.1. **Curry-Howard correspondence.** The Curry-Howard isomorphism describes a correspondence between a given logic and a given programming language. It takes propositions in logic as *types*, proofs as *programs* and normalization of proofs as *evaluation of programs* [Wadler, 2015]. As a consequence, any language designed around these principles provides intrinsic means for reasoning around its programs [Caires et al., 2012]. The $\lambda-$calculus was developed by Alonzo Church as a pure calculus for functional computation, thus it is powerful enough to exhibit any "effectively calculable" function of numbers [Wadler, 2015]. However, given that as a logic, $\lambda-$calculus is inconsistent, Church introduced the *simply-typed* $\lambda-$*calculus*. In this way, the Curry-Howard correspondence could be established between *simply typed* $\lambda-$*calculus* and logic (in natural deduction). For academics working on research in concurrency, it was essential that a foundation as firm as that of $\lambda-$calculus could be developed for concurrency as well [C. Pierce, 1995]. When Jean-Yves Girard pioneered *linear logic* in 1987, a promise of a foundation for concurrency rooted in Curry-Howard emerged [Wadler, 2014]. Linear logic is considered the logic of resources [Braüner, 1996], in which every formula can be used exactly once. In intuitionistic logic, the rule of *the excluded middle* does not hold; thus it is not possible to state that if the negation of a formula does not hold in an environment, then the formula itself holds. In sequent style intuitionistic logic, every sequent has at most one formula in the right-hand side. A long line of research surfaced, centered around the relationship between intuitionistic linear logic and $\pi$-calculus. When session types were introduced, a correspondence between $\pi$-calculus and linear logic was established, where propositions are taken as *session types*, proofs as *processes*, and cut-elimination as *communication* (between processes) [Wadler, 2014]; CPT is a variant of this correspondence.

4.2. **What is CPT.** CPT is a type system where the types of channels are based on the logical connectives of linear logic. By determining the types of channels of a certain process, we are able to discern the behaviour of that process as well. Based on the typing rules that Caires, Pfenning and Tonihno have introduced in their paper [Caires et al., 2012], it is possible to infer whether a process is well-typed in CPT or not.

A process that is well-typed in CPT has a complete derivation tree that can be built using the typing rules of CPT. A derivation tree in CPT is complete if and only if all of its leaves are *axioms*, that is, statements that are taken to be true. Each one of the typing rules in CPT is analogous to the derivation rules of sequent style, intuitionistic linear logic. Before we delve into the typing rules that are relevant to this paper, we discuss the structure of a process in CPT.

4.3. **Processes in CPT.** In the context of logic, a *judgement* of the form $M{:}A$ states that there is a proof $M$ for $A$. In the context of programming languages, we would state that $M$ is a variable of type $A$. Finally, in the context of CPT, such a judgement is of the form $P :: x{:}A$ and states that process $P$ *offers* a *service* of *type* $A$ along channel $x$. However, other than offering services, a process in CPT is able to *use* them as well. The ability of a process to offer and use services is made possible through its own channels. Judgements are used to describe the type of every channel for a particular session of the process. Therefore, by determining the type of every channel, we determine the type of the process as well. Judgments placed on the right-hand side of a process $P$ are used to state the type of channels that communicate services which are offered by $P$; the judgments on the left-hand side of a process $P$ are used to state the type of channels that communicate services that are used by $P$. The entirety of the left-hand side judgments makes up the *context* of a process. There are two different variants of contexts in CPT: *linear context* and *shared context*. We will firstly present the typing rules solely with linear contexts, and further ahead introduce the linear context as well. A sequent in intuitionistic linear logic has the following form:

$$\Gamma \vdash A$$

In CPT, such a sequent is used as the base for the definition of a process:

$$\begin{aligned} x_1{:}A_1, ..., x_n{:}A_n \vdash P :: x{:}A \\ \Leftrightarrow \Gamma \vdash P :: x{:}A \ \text{ where, } \ \Gamma = x_1{:}A_1, ..., x_n{:}A_n \end{aligned}$$

$P$ offers service $A$ along channel $x$, and makes use of services $A_1$, $A_2$ ... $A_n$ along channels $x_1, x_2$, . . . $x_n$. Analogous to sequent linear logic, we refer to the left-hand side channels as *antecedents* and the right-hand side channels as *succedents*. Analogous to sequent intuitionistic logic, there is only one succedent to the right-hand side of a process.
To summarise, a process in CPT is made up of channels that allow it to interact with other processes (the environment). Given that every channel is defined by a type that describes the kind of service that it can communicate, and given that channels are responsible for allowing a process to communicate, types allow us to precisely discern the behaviour of any process.

A process evolves when the channels interact with the environment. Given the specific type of a channel, an interaction implies the behaviour that is specified by the type in question. Once a channel has interacted with the environment, it changes its state and can no longer proceed with the same interaction. A change in state implies a change in type, and therefore, the channel will depict a different behavior. This condition results from the "constraint of resources" principle of linear logic. In this paper, we study the types of channels that communicate the services of *input, output, replication* and *inert*. This is analogous to saying that we study the input, output, replication and inert channel types.

4.3.1. *Communication between processes.* Before we delve into the discussion about channel types in CPT, we consider how two processes interact with one another in CPT. The Curry-Howard isomorphism takes cut-elimination as communication between processes. However, in this paper, we restrict ourselves solely to the *cut* rule from Sequent Calculus. This rule makes it possible to remove redundant formulas from sequents. In a two-sequent premise, if the antecedent $A$ of one sequent is the succedent $A$ of the other, then we can safely remove $A$ from both sequents by applying the *cut* rule:

$$\frac{\Delta_1 \vdash A \qquad \Delta_2, A \vdash \Gamma}{\Delta_1, \Delta_2 \vdash \Gamma}\ \text{cut}$$

In CPT, we substitute the antecedent formulas for the antecedent channels and the succedent formula on the right, for the succedent channel. Consequently, we have:

$$\frac{\Delta_1 \vdash P :: x{:}A \qquad \Delta_2, x{:}A \vdash Q :: z{:}C}{\Delta_1, \Delta_2 \vdash (\boldsymbol{\nu} x)(P \mid Q) :: z{:}C}\ \text{cut}$$

Given two processes $P$ and $Q$, if $Q$ is using channel $x$ that communicates a particular service $A$, and $P$ is offering service $A$ along channel $x$, then $P$ and $Q$ can communicate with one another along channel $x$ while running simultaneously. Channel $x$ is now private to both processes.

4.4. **Channel Types and Typing Rules.** Every channel type in CPT is analogous to a logical connective, and the typing rules are very similar to the application rules of the logical connective. In sequent style intuitionistic linear logic, every connective has a left-hand side typing rule - when it is part of an antecedent - and a right-hand side typing rule - when it is part of a succedent. We refer to these rules as the *left* and *right* rule respectively. Similarly, we can apply the left rule or the right rule to a process, based on whether the channel we are considering is an antecedent or succedent, respectively. We now discuss each one of the channel types separately. For every type:

(1) we will present the logical connective that the channel type is based on,
(2) we will analyze what it means for a process to be able to offer and use a particular channel that communicates a certain service,
(3) we will describe the left and right rules that are applicable.

From now on, when talking about logic, we are referring to sequent style, intuitionistic linear logic.

4.4.1. *Input.* In CPT, the input channel type is based on linear implication. In logic, $A \multimap B$ holds if B is true under the assumption that A is true:

$$\frac{\Delta, A \vdash B}{\Delta \vdash A \multimap B} \multimap R \quad (1)$$

In CPT, $A \multimap B$ is the input type of a succedent channel. In other words, such a channel allows a process $P$ to offer a service of type $A \multimap B$. Given that channels which a process gets to offer are placed to the right-hand side of a process, the right rule is used. When we translate (1) into CPT, we obtain the following derivation tree:

$$\frac{\Delta, y{:}A \vdash P :: x{:}B}{\Delta \vdash x(y).P :: x{:}A \multimap B} \multimap R$$

This states that, if process $P$ offers (some service) $B$ along $x$ when provided with the opportunity to use (some service) $A$ along $y$, then $P$ is able to input $A$ and then proceed as $B$. If we look at the tree from bottom up, what we see is a *session*, where a process $P$ inputs an $A$ and then continues behaving like $B$. In this session, channel $x$ changed state from one type, $A \multimap B$, to the other, $B$.

***How can a process use a channel of the input type?***

We now discuss the opposite situation, where a process uses a channel that has the input type. The idea is to think of how two processes would communicate in a certain situation. In real life, if someone is willing to read from a buffer, then there is someone willing to write to that buffer. In CPT, if a process offers input along channel $x$, this means that another process is providing output along that same channel. If we want to apply the *cut* rule and have the two processes communicate with one another, $x$ needs to have the same type in both processes. Of course, the semantics of the types will be different for each process. Therefore, the type of a channel that is used as the

communicating link between two processes, has opposite meanings when used as an antecedent compared to when it is used as a succedent. Based on this insight, the type of an input antecedent in CPT is identical to the type of an output succedent. The two types are based on the logical connective that is called *tensor* ($\otimes$), and its left rule is defined by the tree:

$$\frac{\Delta, A, B \vdash C}{\Delta, A \otimes B \vdash C} \otimes\text{L (2)}$$

When we translate (2) into CPT, we have the following derivation tree:

$$\frac{\Delta, y{:}A, x{:}B \vdash Q :: z{:}C}{\Delta, x{:}A \otimes B \vdash x(y).Q :: z{:}C} \otimes\text{L}$$

After $y$ of type $A$ is received along $x$, the later changes state and becomes of type $B$. Both channels can be used for communication with other processes that offer types $A$ or $B$.
To conclude, when a process offers an input service, it does so using a channel of the input type $A \multimap B$; when the process uses a channel of the input type, the type is equal to $A \otimes B$. In both cases, service $A$ is input along the channel and then the channel switches to type $B$.

4.4.2. *Output.* In CPT, the output service is a form of bound output. Thus, the message that will be sent along a channel is private to two processes running concurrently. The output type of a succedent channel is $A \otimes B$. In logic, if it is possible to prove A and B separately, then it is possible to prove A $\otimes$ B as well. The right rule for $\otimes$ is:

$$\frac{\Delta_1 \vdash A \qquad \Delta_2 \vdash B}{\Delta_1, \Delta_2 \vdash A \otimes B} \otimes\text{R (3)}$$

In CPT, we would translate (3) into the following derivation tree:

$$\frac{\Delta_1 \vdash P :: y{:}A \qquad \Delta_2 \vdash Q :: x{:}B}{\Delta_1, \Delta_2 \vdash (\boldsymbol{\nu} y)\overline{x}\langle y\rangle.(P \mid Q) :: x{:}A \otimes B} \otimes\text{R}$$

In this session, $x$ is of type $A \otimes B$. Once a process outputs $y$ along $x$, we end up with two new processes, where one offers $y$, which now has type $A$ and the other one offers $x$ which changes state to type $B$. Of course, given that $x$ is a succedent, the right rule is applied.

***How can a process use a channel of the output type?***
To use a channel of the output type, it means that another process must input along the same channel. Therefore, the type of an output antecedent is similar to the type of an input succedent, that is $A \multimap B$. In this situation, we apply the left rule for the logical connective $\multimap$:

$$\frac{\Delta_1 \vdash A \qquad \Delta_2, B \vdash C}{\Delta_1, \Delta_2, A \multimap B \vdash C} \multimap\text{L}$$

If we translate this rule to CPT, then we have the following derivation tree:

$$\frac{\Delta_1 \vdash P :: y{:}A \qquad \Delta_2, x{:}B \vdash Q :: z{:}C}{\Delta_1, \Delta_2, x{:}A \multimap B \vdash (\boldsymbol{\nu} y)\overline{x}\langle y\rangle.(P \mid Q) :: z{:}C} \multimap\text{L}$$

4.4.3. *Inert.* In CPT, the inert type is used to state that a process is not interacting with the environment. We use **1** to define such a type, and there are two rules as well. From the logical rules:

$$\frac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}\text{R} \quad \text{and} \quad \frac{\Delta \vdash C}{\Delta, \mathbf{1} \vdash C}\ \mathbf{1}\text{L}$$

we obtain the following derivation trees in CPT respectively:

$$\frac{}{\cdot \vdash \mathbf{0} :: x{:}\mathbf{1}}\ \mathbf{1}\text{R (4)} \quad \text{and} \quad \frac{\Delta \vdash P :: z{:}C}{\Delta, x{:}\mathbf{1} \vdash P :: z{:}C}\ \mathbf{1}\text{L (5)}$$

In (4), we see that an inert process has no antecedents. This process offers nothing, as stated by the type of channel $x$. The application of this typing rule renders axioms. In (5), we note that an antecedent of type **1** can be removed from the context of the sequent because it serves no purpose whatsoever.

4.4.4. *Replication.* In real life, a server is repeatedly waiting for requests from its clients. Once received, the server responds to every request separately, while still waiting for other ones. In CPT, this behavior is identifiable with the replication process. Let us recall from reduction under the condition of replication [§3.3] that:

$$(\boldsymbol{\nu} x)(!x(z).P \mid (\boldsymbol{\nu} y)\overline{x}\langle y\rangle.Q) \rightarrow (\boldsymbol{\nu} y)(P\{y/z\} \mid (\boldsymbol{\nu} x)(!x(z).P \mid Q))$$

If we look at the right hand side, $P\{y/z\}$ represents the server's response to the client's request and $!x(z).P$ represents its resumption to accepting new requests. Such behaviour is made possible by channel $x$ that awaits for input. We refer to this channel as a *server channel.* Given the relevance of this process, it is understandable that in a concurrent type system, there is a type for a server channel. Such a type is identified by placing a ! *(bang)* ahead of the server channel. Therefore, in $!x(y).P$, channel $x$ is constantly waiting for input, and once received in the form of $z$, a copy of $P$ will run. At the same time, $x$ will wait for other inputs in the form of $z$. Consequently, we state that a server channel represents the offer of a replicated input. In such a session, $x$ does not change state, in that, it will always remain a server channel. However, this idea counters the "constraint of resources" idea that CPT is based on. As a result, we now introduce the concept of *shared context.*

Every typing rule that we have considered thus far has been analogous to some particular sequent calculus rule. The context of every process in CPT has been *linear*, in that, every antecedent could be used exactly once. Therefore, a channel that does not change state cannot be placed in such a context. Instead, we make use of the two variants of contexts introduced in §4.3: one context where we define the judgments for the shared channels (shared context) and the other where we define the judgments for the linear channels (linear context). A judgment for the server channel will, then, be placed in the shared context. Given these insights, we may now introduce a new typing rule, called the *cut!* rule. The rule in logic:

$$\frac{\Gamma; \cdot \vdash A \qquad \Gamma, A; \Delta \vdash C}{\Gamma; \Delta \vdash C}\ \text{cut!}$$

translates into the following derivation tree in CPT:

$$\frac{\Gamma; \cdot \vdash P :: x{:}A \qquad \Gamma, u{:}A; \Delta \vdash Q :: z{:}C}{\Gamma; \Delta \vdash (\boldsymbol{\nu} u)(!u.(x).P \mid Q) :: z{:}C}\ \text{cut!}$$

In this rule, we have two different contexts. We distinguish between the two by separating them with a semi-colon. First, we present the shared context, and then the linear context. Every typing rule that has been introduced thus far can be adapted so that the shared context is included as

well. No further changes in the application of the rule are required. For example, the $\mathbf{1}R$ rule now becomes:

$$\frac{}{\Gamma; \cdot \vdash \mathbf{0} :: x{:}\mathbf{1}}\ \mathbf{1}\mathrm{R}$$

The *cut!* rule states that, given a process $P$ without any linear antecedents, but a succedent $x$ of type $A$, and a process $Q$ with a persistent antecedent $u$ of type $A$, a new process composed of both $P$ and $Q$ can be formed, with $u$ being a server channel that receives input of type $A$.

***How do we instigate a server channel?***
For every server that is persistently waiting for input, there must be a process that sends some bound output along the server channel. We denote this situation with *copy* rule:

$$\frac{\Gamma, u{:}A; \Delta, y{:}A \vdash P :: z{:}C}{\Gamma, u{:}A; \Delta \vdash (\boldsymbol{\nu} y)\overline{u}\langle y\rangle.P :: z{:}C}\ \mathrm{copy}$$

If a process $P$ has two channels of the same type, one in the linear context, and one in the shared context, then we can remove the linear antecedent and use it as a message to communicate along the shared channel. We now introduce the left and right rules that are applied to a server channel when it is an antecedent and a succedent, respectively. First, we consider the right rule.

$$\frac{\Gamma; \cdot \vdash P :: y{:}A}{\Gamma; \cdot \vdash\ !x(y).P :: x{:}!A}\ !\mathrm{R}$$

According to this rule, a process that offers the service of a replicated input, has a strictly empty, linear context. We now consider the left rule for *bang*:

$$\frac{\Gamma, u{:}A; \Delta \vdash Q :: z{:}C}{\Gamma; \Delta, x{:}!A \vdash \{x/u\}.Q :: z{:}C}\ !\mathrm{L}$$

This rule upgrades a channel from linear to shared status without changing the semantics of process $Q$. If we look at the rule from top to bottom, every occurrence of $u$, gets substituted for $x$.

4.5. **Session Types.** Following the types and the typing rules that we introduced previously, we note that every channel gets to act according to a certain type exactly *once*. Should that same channel be used again in the process, then it will behave according to a different type. This notion is encapsulated by the concept of *session types*. That is, a particular session causes a channel to change from one state to another. Each one of the types that we introduced describe a particular session. For example, the type $A \multimap B$ describes the type of a session that inputs an $A$ and then behaves like $B$.

Having introduced the session types, we may now conclude the section of this paper that is dedicated to CPT. We discussed four types of channels. Types describe the behaviour of a channel. In CPT channels take two different stands: either to the left of the process or to its right. This matter requires for the application of separate typing rules that are concerned with the specific type of the channel. Furthermore, there are also rules that do not pertain to a specific channel type, however, allow for the evolution of the processes. The typing rules allow us to analyse the structure of a process. Once we have applied the ***1R*** rule, we have reached the end of the process and we can conclude that the process in question is well-typed in CPT. *A process that is well-typed in CPT, is strongly normalizing*, that is, it terminates. We now introduce the second type system.

## 5. The DS Type System

Yuxin Deng and Davide Sangiorgi have introduced four, concurrency-related, type systems in their paper, "Ensuing termination by typability". Three of them are refinements of one another, where a newer version builds on the restrictions of a previous one. Only the first type system, namely, *the core system* [Deng and Sangiorgi, 2006], is relevant to this thesis. Throughout the rest of the chapters, we will refer to this type system as DS. Any process that is well-typed in DS, is strongly normalizing [Deng and Sangiorgi, 2006]. In DS, there are several *typing rules* that can be used to construct derivation trees. Similar to CPT, derivation trees allow us to conclude whether a process is well-typed in DS, therefore, we can determine whether the process terminates. In the following subsections, we will discuss the basic concepts of DS that are relevant to this paper, how channel types are defined in DS and how the typing rules are used to construct the derivation trees.

### 5.1. **Quick introduction of the basic concepts.**

5.1.1. *Types.* In DS, a *channel* is a name that could be used to engage in communications [Deng and Sangiorgi, 2006]. The terms channel and *variable* are used interchangeably. Types in DS are defined so that they can be used to describe the values of the variables that can be transmitted in communications. When channels engage in a communication, they exchange objects with one another. These objects can be either channels themselves, or simple variables such as an integer. Therefore we can define two different variable types in DS: *link types* and *basic types*. Both types are denoted with upper-case letters of the alphabet such as $M, N, L, K...T$.

Link types represent the channels. That is, they are used to define the type of variables that can carry other link and basic type variables as well. The typing rules of DS are given only for link types. Link types are distinguishable from basic types because they are denoted with a # (*pound*). The pound symbol is also used to state the level of such a channel; the domain of a level is the set of all natural numbers. As an example, we consider: $x : \#^3 T$, where channel $x$ has link type $T$, and level 3. This means that channel $x$, with level 3, can carry other link type variables (or simply channels) of type $T$.

Basic types are used to carry only *boolean* and *numerical values*, and are unable to carry other types. Therefore, in DS, any transmittable variable, can take the following values:

$$u, w ::= x \mid true, false \mid 0, 1, 2 \ \ ...$$

where, $x$ is a channel

5.1.2. *Judgments.* The concept of a judgment in DS is different from the concept in CPT, because DS is not based on a Curry-Howard foundation. Both of the following statements are judgements in this thesis:

$$x : A \multimap B \quad \text{and} \quad \vdash P.$$

However, they describe different situations in different type systems. The first one states the type of channel $x$, but the second one states that process $P$ is well-typed in DS, and thus, strongly normalizing. We will note further ahead that judgments are especially useful when constructing derivation trees in DS: they help us denote the current state of a process and which typing rule is

relevant for the next step. A derivation tree in DS is complete when every channel type has been determined. The leaves of a complete derivation tree are empty, and we refer to them as *axioms*.

5.1.3. *os(P).* Another relevant concept for this paper is the function $os(P)$, $P$ being an arbitrary process. We use this function to state the set of all the output subjects that are active in $P$. An output subject is active in a process if it is not underneath any replicated input [Deng and Sangiorgi, 2006]. Given two processes:

$$\overline{b}y.P \quad \text{and} \quad !a(x).\overline{b}y.P,$$

channel $b$ is an active output subject in the first one, but not in the second one. Therefore, the application of $os$ to the first process results in $\{b\}$, whereas to the second one in $\emptyset$.

5.1.4. *Processes in DS.* We restrict ourselves to the same subset of processes in DS that were introduced in §3.1. We introduce two subtle syntactic differences with the processes of CPT:

(1) With regards to the restriction process, in DS we use $\boldsymbol{\nu} aP$ to denote that channel $a$ is private to process $P$, as opposed to $(\boldsymbol{\nu} a)$P in CPT.
(2) In DS, the output process is not bound, as opposed to CPT.

5.2. **Typing rules in DS.** The typing rules in DS are similar to the typing rules in CPT, in that they help us discern the behaviour of the process by considering the behaviour of every channel in the process. However, the typing rules themselves are different for the two type systems. In DS, the typing rules coincide with the types of a process. We consider exactly *six* rules which correlate precisely with the six processes introduced in §3.1. Every rule pertains to a derivation tree where the lower part is the current state of the process in question, which we assume is well-typed in DS. The upper part of the tree is the new state of the process after the application of the relevant typing rule, stated to the right of the process. Every rule application renders the process smaller and allows us to discern the types of every channel. We now introduce the rules accordingly.

(1) *T-in:*

$$\frac{a : \#T \;\; x : T \;\; \vdash P}{\vdash a(x).P}\ \text{T-in}$$

This rule allows for the typing of a channel $a$ that accepts input $x$. By assuming that $a(x).P$ is well-typed in DS, *T-in* entitles channel $a$ to having an arbitrary link type $T$, input $x$ to having type T, and the rest of the process, namely P, to be well-typed in DS.

(2) *T-out:*

$$\frac{a : \#T \;\; v : T \;\; \vdash P}{\vdash \overline{a}v.P}\ \text{T-out}$$

This rule allows for the typing of the output process. Channel $a$ outputs variable $b$ and as such, is entitled to an arbitrary link type $T$, $v$ is entitled to type $T$ and process $P$ is well-typed in DS.

(3) *T-nil:*

$$\frac{}{\vdash \mathbf{0}}\ \text{T-nil}$$

This rule is used to denote that the process has come to its end. There are no more channels to incite communication with other processes. Thus, the upper part of the tree is empty. The result of applying this rule while building a derivation tree, is an axiom.

(4) *T-par:*

$$\frac{\vdash P \quad \vdash Q}{\vdash P \mid Q}\ \text{T-par}$$

This rule is designated to type a process that is composed of two sub-processes running concurrently. By assuming that $P \mid Q$ is well-typed in DS, *T-par* allows us to conclude that $P$ and $Q$ are also well-typed in DS.

(5) *T-res:*

$$\frac{a : L \quad \vdash P}{\vdash \boldsymbol{\nu} aP}\ \text{T-res}$$

The restriction rule allows us to type a process with a private variable. If we assume that $\boldsymbol{\nu} aP$ is well-typed in DS, then so is $P$. Therefore, $\boldsymbol{\nu} aP$ terminates if and only if $P$ terminates as well.

(6) *T-rep:*

$$\frac{a : \#^n T \quad x : T \quad \vdash P \quad \forall b \in os(P),\ \ lv(b) < n}{\vdash !a(x).P}\ \text{T-rep}$$

The replication rule allows us to type a process of the replicated input. It states that if $!a(x).P$ is well-typed in DS, then $a$ has a link type $a : \#^n T$, the input $x$ has type $T$, and every other active output subject in $P$ has a level that is strictly smaller than the level of $a$. This is a very important rule in DS as it ascertains that a process which showcases the behavior of replicated input will always terminate. Given the following process:

$$P \stackrel{\text{def}}{=} !a.(\overline{c} \mid !b.\overline{a}),$$

applying the *T-rep* rule requires the calculation of $os(\overline{c} \mid !b.\overline{a})$, the set of all active output subjects. Since an output subject is active if it does not occur in the body of a replicated input, channel $c$ is active. The second occurrence of $a$, however, is inactive, because it is bound by the server channel $b$. Thus, the resulting set we obtain is $\{c\}$. Consequently, the level constraints that *T-rep* sets for $P$ are the following:

(i) The level of $c$ needs to be strictly smaller than the level of $a$.
(ii) The level of $a$ needs to be strictly smaller than the level of $b$: when we apply *T-rep* to $!b.\overline{a}$, we have to calculate $os(\overline{a})$, which results in $\{a\}$.

Given this insight, process $P$ is well-typed in DS, if and only if constraints (i) and (ii) hold. Thus, a correct typing for the channels of $P$ could be:

$$a : \#^2\mathrm{T} \quad b : \#^3\mathrm{T} \quad c : \#^1\mathrm{T}$$

In DS, when the type of a channel is defined due to a typing rule, it could be the case that it is not final, because another rule will elaborate on it. The type of the channel can be determined when the derivation tree is complete and will be the one that encompasses all the behaviours that the channel allows for. This type encloses the type(s) that were introduced beforehand as well.
Having introduced a few of the basic concepts and the typing rules, we may now conclude our introduction to the DS type system. A channel in DS either has a link type or a basic type; the typing

rules allow us to determine the types for every channel. When we are constructing a derivation tree in DS, it is always possible to apply one of the six typing rules, because there are no constraints placed on the rules. However, the *T-rep* rule does place constraints on the levels of channels. In order for a process to be well-typed in DS, such constraints need to be satisfied. *A process that is well-typed in DS, is strongly normalizing*: if $\vdash P$ then $P$ terminates. We will now commence a comparison between the two type systems, CPT and DS. We will present three different processes and showcase how they can be typed in each type system.

## 6. Comparing the two Type Systems

In this section, we will present four processes; three out of which are well-typed in DS, but not in CPT, and a fourth one that is well-typed in DS and CPT. We will present a thorough analysis of every process and provide an outlook on why it is the case that they can or cannot be well-typed in CPT. We firstly introduce the processes with the syntax of DS, before delving into each one of them separately:

(i) $P_1 \stackrel{\text{def}}{=} a(x).!x(y).\overline{c}d \mid !c(b).\overline{b}e$
(ii) $P_2 \stackrel{\text{def}}{=} \overline{b}a.!a(y).\overline{y}t \mid !b(x).\overline{x}z \mid !z(m).m(u)$
(iii) $P_3 \stackrel{\text{def}}{=} !a(y).z(t) \mid !b(x).x(y) \mid \overline{a}y \mid \overline{b}x$
(iv) $P_4 \stackrel{\text{def}}{=} \overline{a}x.x(y) \mid !a(b).\overline{b}c$

Process $P_1$ represent two sub-processes running in parallel. There are two servers, however there are no clients to activate either one of the servers. On the left side, server channel $x$ is received along channel $a$ and then channel $c$ is used to output some channel $d$. On the right side, server channel $c$ accepts channel $b$ as its input and then uses it to transmit some message $e$.

Process $P_2$ represents three sub-processes running in parallel, with all three containing server channels. The first sub-process is responsible for activating server $b$ on the second sub-process. Server $b$ is then used to activate server $a$ on the first sub-process. Finally, $a$ is used to activate server $z$ on the third sub-process. After activation, each server either inputs or outputs some channel.

Process $P_3$ represents four sub-processes running in parallel, where the first two contain servers. The other two are used to activate the servers. No communication between the servers themselves occurs. After activating the servers, the processes either input or output some channel.

Process $P_4$ represents two sub-processes running in parallel, where one of the sub-processes is used to activate the server on the other one. Therefore, we have a client sub-process on the left, and a server sub-process that responds to the client on the right. We now begin with the analysis.

### 6.1. **First process.**

**Lemma 6.1.** $P_1$ is well-typed in DS, but not in CPT.

*Proof.* We firstly introduce process $P_1$ according to the syntax of DS. Then, we present the complete derivation tree for its type system and reason about the concluded channel types. Afterwards, we present $P_1'$ which, for simplicity, we assume that it is analogous to $P_1$ in CPT. We then show that it is not possible to construct a complete derivation tree for $P_1'$ in CPT with a proof by contradiction.

6.1.1. *Typability in DS.* Firstly, we show the construction of the derivation tree in DS for $P_1$, using the typing rules from §5.2, where: $P_1 \stackrel{\text{def}}{=} a(x).!x(y).\bar{c}d \mid !c(b).\bar{b}e$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{c : \#^m T' \;\; d : T' \;\; \vdash \mathbf{0}}\text{T-nil}}{x : \#^n T'' \;\; y : T'' \;\; \vdash \bar{c}d \quad lv(c) < n}\text{T-out}}{a : \#T \;\; x : T \;\; \vdash !x(y).\bar{c}d}\text{T-rep}}{\vdash a(x).!x(y).\bar{c}d}\text{T-in} \qquad \dfrac{\dfrac{\dfrac{}{b : \#T''' \;\; e : T''' \;\; \vdash \mathbf{0}}\text{T-nil}}{c : \#^m T' \;\; b : T' \;\; \vdash \bar{b}e \quad lv(b) < m}\text{T-out}}{\vdash !c(b).\bar{b}e}\text{T-rep}}{\vdash a(x).!x(y).\bar{c}d \mid !c(b).\bar{b}e}\text{T-par}$$

NOTE: For styling reasons, we assume that every channel type is carried on from each lower branch to an upper one.

REMARKS:

The derivation tree is complete as every leaf of the tree is an axiom resulting from the application of *T-nil*. This tree is made out of two sub-trees, which we assume are built simultaneously. When a channel occurs more than once, and upon a new occurrence, the type of a channel has already been determined and can represent the new behaviour, it will remain the same. Since channel $c$ is common to both sub-trees, the type resulting from the application of *T-rep* is the final one. Channel $c$ first appears in the left sub-tree and further in the right sub-tree. In both cases, the same type is used. We conclude that $P_1$ is well-typed in DS, given the following typing for each link type channel:

$$a : \#T \;\; x : \#^n T'' \;\; c : \#^m T' \;\; b : \#T''' \text{ and } lv(b) < m, m < n$$

Since $P_1$ can be well-typed in DS, then we can conclude that $P_1$ is strongly normalizing.

6.1.2. *Typability in CPT.* We now consider $P_1'$, which we assume is the analogous procces of $P_1$ in the syntax of CPT. By assuming that $P_1'$ is well-typed in CPT, we may build a derivation tree using the typing rules introduced in §4.4. Let us first consider the process:

$$P_1' \stackrel{\text{def}}{=} (\boldsymbol{\nu} c)(a(x).!x(y).(\boldsymbol{\nu} d)\bar{c}\langle d\rangle.(\mathbf{0} \mid \mathbf{0})) \mid !c(b).(\boldsymbol{\nu} e)\bar{b}\langle e\rangle.(\mathbf{0} \mid \mathbf{0})$$

We note the following:

(i) The only free channel in this process is $a$. Every other channel is either bound ($x$ and $b$), or we cannot make any statements about their type ($c$).

(ii) As a consequence of (i), we can only create a judgment for channel $a$, which receives input $x$ and then does not perform anything else.

(iii) Given point (ii), we will consider two cases for proving the typability of $P_1'$ in CPT:
- Case 1, where we regard $a$ as an antecedent and thus place its judgement to the left of the process. In this case, a dummy variable $f{:}1$ which shows no behavior in the process, will be used as the succedent.
- Case 2, where we regard $a$ as the succedent. Consequently, the linear context will initially be empty. We assume the shared context to be $\Gamma$.

(iv) $P_1'$ is a parallel composition of two sub-processes. In CPT, two parallel processes that have a channel in common can communicate with one another by means of the *cut* rule. Since $c$ is shared by both sub-processes, we now consider the type of the channel, which has to be the same for both sub-processes. Given the second sub-process:

$$\text{(a)} \quad !c(b).(\boldsymbol{\nu} e)\bar{b}\langle e\rangle.(\mathbf{0} \mid \mathbf{0}),$$

it is easy for us to note that $c$ is a server channel, and as such, offers the service of a replicated input. Therefore, sub-process (a) offers $c{:}!A$. Consequently, the other sub-process:

$$\text{(b)} \quad a(x).!x(y).(\boldsymbol{\nu} d)\overline{c}\langle d\rangle.(\mathbf{0} \mid \mathbf{0})$$

has to use channel $c$, that is, $c{:}!A$ is an antecedent of (b).

Based on the aforementioned insights, we now construct the derivation trees for both cases from (iii). Unlike the derivation tree for $P_1$ in DS, the sub-trees in CPT are not dependent on one another. Thus, we first construct the left sub-tree and afterwards, the second sub-tree.

## Case 1

We now showcase the derivation tree that can be constructed from Case 1:

$$\dfrac{\dfrac{\dfrac{}{\Gamma;\cdot \vdash \mathbf{0} :: e{:}1}\,1\text{R} \quad \dfrac{}{\Gamma;\cdot \vdash \mathbf{0} :: b{:}1}\,1\text{R}}{\Gamma;\cdot \vdash (\boldsymbol{\nu} e)\overline{b}\langle e\rangle.(\mathbf{0} \mid \mathbf{0}) :: b{:}A}\,\otimes\text{R},\ *^1}{\Gamma;\cdot \vdash !c(b).(\boldsymbol{\nu} e)\overline{b}\langle e\rangle.(\mathbf{0} \mid \mathbf{0}) :: c{:}!A}\,!\text{R} \qquad \dfrac{\dfrac{\dfrac{\Gamma, u{:}D; c{:}!A \vdash !u(y).(\boldsymbol{\nu} d)\overline{c}\langle d\rangle.(\mathbf{0} \mid \mathbf{0}) :: f{:}1}{\Gamma, u{:}D; a{:}1, c{:}!A \vdash !u(y).(\boldsymbol{\nu} d)\overline{c}\langle d\rangle.(\mathbf{0} \mid \mathbf{0}) :: f{:}1}\,1\text{L}}{\Gamma; a{:}1, x : B, c :!A \vdash !x(y).(\boldsymbol{\nu} d)\overline{c}\langle d\rangle.(\mathbf{0} \mid \mathbf{0}) :: f{:}1}\,!\text{L},\ *^2}{\Gamma; a{:}B \otimes 1, c :!A \vdash a(x).!x(y).(\boldsymbol{\nu} d)\overline{c}\langle d\rangle.(\mathbf{0} \mid \mathbf{0}) :: f{:}1}\,\otimes\text{L}$$

$$\Gamma; a{:}B \otimes 1 \ \vdash (\boldsymbol{\nu} c)(a(x).!x(y).(\boldsymbol{\nu} d)\overline{c}\langle d\rangle.(\mathbf{0} \mid \mathbf{0})) \mid !c(b).(\boldsymbol{\nu} e)\overline{b}\langle e\rangle.(\mathbf{0} \mid \mathbf{0}) :: f{:}1 \quad \text{cut}$$

$*^1$: where, $A = 1 \otimes 1$

$*^2$: where, $B = !D$

REMARKS:

(i) In $*^1$, we need to apply the rule that concerns the type of channel $b$. At this point, we can discern that $b$ provides a new, bound output and as such, we can infer that the type of this session is $1 \otimes 1$, because both $b$ and $e$ will behave as the inert process afterwards.

(ii) In $*^2$, the next session depends on the type of channel $x$. Upon being received along $a$, $x$ now acts as a server channel. For this reason, we may infer that the type of $x$ is $!D$, $D$ being an arbitrary session type that we can deduce further in the process.

(iii) We note that the left sub-tree is complete: the process terminates with $\mathbf{0}$ and both its leaves are axioms. Consequently, we can conclude the types of channels at the beginning of the session:

$$a{:}1 \ \text{ and } \ b{:}(1 \otimes 1)$$

(iv) The right sub-tree is not complete, as the leaf is not an axiom. The current session:

$$\Gamma, u{:}D; c{:}!A \vdash !u(y).(\boldsymbol{\nu} d)\overline{c}\langle d\rangle.(\mathbf{0} \mid \mathbf{0}) :: f{:}1$$

consists of a server channel $u$, that accepts input $y$ and then allows for channel $c$ to output a new $d$. There is solely one antecedent present, which describes the behavior of channel $c$. The only way to be able to get to the session of channel $c$, is by applying *!R*. The application of *!R* requires that the server channel be a succedent of the current session and that no antecedents be present in the session. However, server channel $u$ is not the succedent of the session and the application of *!L* beforehand, updated its status to a shared channel. In CPT, the typing rules for connectives apply only to channels that are defined in the linear context.

Given the remarks, we conclude that it is not viable to construct a complete derivation tree in CPT for Case 1, and proceed with Case 2.

## Case 2

We will now consider the second situation, where we decide to place the judgement over channel $a$ as a succedent. Since there are no differences with regard to the left sub-tree of Case 1, we decide to showcase only the right sub-tree for Case 2.

$$\dfrac{\dfrac{\Gamma, u{:}D; c{:}!A \quad \vdash !u(y).(\boldsymbol{\nu} d)\overline{c}\langle d\rangle.(\mathbf{0} \mid \mathbf{0}) :: a{:}1}{\Gamma; c{:}!A, x{:}B \quad \vdash !x(y).(\boldsymbol{\nu} d)\overline{c}\langle d\rangle.(\mathbf{0} \mid \mathbf{0}) :: a{:}1}\ !\text{L},\ *}{\Gamma; c{:}!A \vdash a(x).!x(y).(\boldsymbol{\nu} d)\overline{c}\langle d\rangle.(\mathbf{0} \mid \mathbf{0}) :: a{:}B \multimap 1}\ \multimap\text{R}$$

$*$: where B=!D

REMARKS:

(i) In $*$, we need to apply the rule that is relevant to the type of $x$. Since $x$ is a server channel, then we may infer that its type is $!D$, $D$ being an arbitrary type that we can deduce further in the proof.

(ii) We note that the result of this sub-tree is very similar to the right sub-tree of Case 1. Once again, we cannot apply *!R* because the judgment over server channel $u$ is not a succedent. Instead, server channel $u$ was updated to shared status from rule $!L$ and no other rule from §4.4 can be used to continue to the session after the server.

As a consequence of remarks (i) and (ii), we conclude that it is not possible to construct a complete derivation tree using Case 2.

***Conclusion***

Since no complete derivation tree for $P_1'$ could be constructed neither from Case 1 nor from Case 2, we may conclude that our assumption that $P_1'$ is well-typed in CPT, leads to a contradiction. Consequently, $P_1'$ is not well-typed in CPT. As a result, we are able to confirm 6.1. □

Although two servers are present in this process, none of the two is activated by another sub-process. In this paper, we refer to this type of process as *static*. What we discerned from $P_1'$ is that only one of the sub-processes could be typed in CPT. The difference between the two server channels is that in sub-process (i), the server channel is not bound by another one, whereas in sub-process(ii), server channel $x$ is bound by $a$. Next, we will consider two non-static processes, one where the server channels get activated by one another in the form of a pipeline, and another one where none of the servers is bound by another channel.

### 6.2. **Second process.**

**Lemma 6.2.** $P_2$ is well-typed in DS, but not in CPT.

*Proof.* In the first process that we presented, the left rule for the *bang* connective was used, which updated the server channel to *shared context*, and left us with no choice for continuing to the next session. We now consider a new process with three servers, where one server activates another one, which in turn, activates the third one. As such, $P_2$ is a parallel composition of three sub-processes. Given: $P_2 \stackrel{\text{def}}{=} \overline{b}a.!a(y).\overline{y}t \mid !b(x).\overline{x}z \mid !z(m).m(u)$, we consider every sub-process separately:

(i) $\overline{b}a.!a(y).\overline{y}t$

(ii) $!b(x).\overline{x}z$

(iii) $!z(m).m(u)$

(1) Sub-process (i) first activates server $b$ of sub-process (ii) by passing along server $a$. Then server $a$ awaits for input $y$, and upon receiving it, sends $t$ along it.
(2) In sub-process (ii), server $b$ awaits for input $x$, which then uses it to pass $z$. When it runs in parallel with (i), $x$ will be substituted for $a$. Consequently, channel $z$ gets passed along $a$, which then activates the server in (i).
(3) If sub-process (iii) runs in parallel with (i) and (ii), server $z$ gets activated due to server $a$ in (i).

We present the reduction steps for $P_2$ to show that it, indeed, has a non-terminating behaviour:

$$\begin{gathered}\overline{b}a.!a(y).\overline{y}t \mid !b(x).\overline{x}z \mid !z(m).m(u)\\ \longrightarrow\\ !a(y).\overline{y}t \mid !b(x).\overline{x}z \mid \overline{a}z \mid !z(m).m(u)\\ \longrightarrow\\ !a(y).\overline{y}t \mid \overline{z}t \mid !b(x).\overline{x}z \mid !z(m).m(u)\\ \longrightarrow\\ !a(y).\overline{y}t \mid !b(x).\overline{x}z \mid !z(m).m(u) \mid t(u)\end{gathered}$$

The reduction steps allow us to infer that $P_2$ is a process with a non-terminating behaviour, because none of the servers is constantly responding to requests. Instead, they remain available, without wasting resources. We will now present the derivation trees for $P_2$ in DS, then we will introduce $P_2'$, which we deem analogous to $P_2$ in CPT, and present its derivation tree in CPT.

6.2.1. *Typability in DS.* We will first construct the derivation tree for $P_2$ in DS using the typing rules introduced in §5.2.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{m : \#T' \;\; u : T' \;\; \vdash \mathbf{0}}\text{T-nil}}{z : \#^t T \;\; m : T \;\; \vdash m(u)}\text{T-in}}{\vdash !z(m).m(u)}\text{T-rep} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{x : \#M \;\; z : M \vdash \mathbf{0}}\text{T-nil}}{b : \#^h T'' \;\; x : T'' \;\; \vdash \overline{x}z \quad lv(x) < h}\text{T-out}}{\vdash !b(x).\overline{x}z}\text{T-rep} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{y : \#K \;\; t : K \;\; \vdash \mathbf{0}}\text{T-nil}}{a : \#^k N \;\; y : N \;\; \vdash \overline{y}t \quad lv(y) < k}\text{T-out}}{b : \#^h T'' \;\; a : T'' \;\; \vdash !a(y).\overline{y}t}\text{T-rep}}{\vdash \overline{b}a.!a(y).\overline{y}t}\text{T-out}}{\vdash \overline{b}a.!a(y).\overline{y}t \mid !b(x).\overline{x}z}\text{T-par}}{\vdash \overline{b}a.!a(y).\overline{y}t \mid !b(x).\overline{x}z \mid !z(m).m(u)}\text{T-par}$$

NOTE: For styling reasons, we assume that every channel type is carried on from each lower branch to an upper one.

REMARKS:

The derivation tree for $P_2$ is complete as every leaf of the branch is an axiom. $P_2$ is well-typed in DS, given this typing of the link type channels:

$$b : \#^h T'' \;\; a : \#^k N \;\; y : \#K \;\; x : \#M \;\; z : \#^t T \;\; m : \#T', \quad lv(x) < h, \quad lv(y) < k,$$

where $K, M, N, T, T', T''$ are arbitrary link types

Thus, as long as the level of channel $x$ is strictly smaller than the level of server $b$, and the level of channel $y$ is strictly smaller than the level of server $a$, $P_2$ is well-typed in DS. Consequently, since $x$ gets substituted for $a$ in (ii), then the level of channel $a$ is strictly smaller than the level of channel $b$ as well. Similarly, since $y$ gets substituted for $z$ in (i), then the level of $z$ should be strictly smaller than the level of $a$. As a result, $P_2$ is strongly normalizing if and only if condition $lv(z) < lv(a) < lv(b)$ is satisfied.

6.2.2. *Typability in CPT.* We now consider process $P_2'$, which we assume is analogous to process $P_2$ according to the syntax of CPT. By assuming that $P_2'$ is well-typed in CPT, we may construct its derivation tree using the rules in §4.4. $P_2'$ takes the following structure:

$$P_2 \stackrel{\text{def}}{=} (\boldsymbol{\nu} z)((!z(m).m(u).\mathbf{0}) \mid (\boldsymbol{\nu} b)((!b(x).\overline{x}\langle z\rangle.(\mathbf{0} \mid \mathbf{0})) \mid (\boldsymbol{\nu} a)\overline{b}\langle a\rangle.!a(y).\overline{y}\langle t\rangle))$$

We note the following:

(i) The channels that are responsible for the behavior of the process are $z, m, b, x, a$ and $y$.
(ii) Channels $a$ and $x$ are bound by $b$, $y$ is bound by $a$ and $z$ is bound by $x$.
(iii) Although $b$ is a free channel, it showcases two different behaviors in two sub-processes: once as a server channel and once as an output prefix. Therefore, we cannot define only one type for $b$, neither can we apply the *cut* rule. Instead, we will resort to the *cut!* rule.
(iv) Given points (i) and (ii), we infer that $P_2'$ has no linear context. Therefore, we will use a dummy variable $d{:}1$ as its succedent. We assume the shared context to be $\Gamma$.

Due to our reasoning [(i)-(iv)], we present the following derivation tree for $P_2'$ in CPT :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma;\cdot \vdash \mathbf{0} :: m{:}1}\,1\text{R}}{\Gamma; u : 1 \vdash \mathbf{0} :: m{:}1}\,1\text{L}}{\Gamma;\cdot \vdash m(u).0 :: m{:}A}\,{\multimap}\text{R},\,*^1 \qquad \dfrac{\Gamma, z{:}A;\cdot \vdash \overline{x}\langle z\rangle.(\mathbf{0} \mid \mathbf{0}) :: x{:}B^{*^2} \qquad \Gamma, z{:}A, b{:}B;\cdot \vdash (\boldsymbol{\nu} a)\overline{b}\langle a\rangle.!a(y).\overline{y}\langle t\rangle :: d{:}1^{*^3}}{\Gamma, z{:}A;\cdot \vdash (\boldsymbol{\nu} b)((!b(x).(\boldsymbol{\nu} n)\overline{x}\langle n\rangle.(\mathbf{0} \mid \mathbf{0})) \mid (\boldsymbol{\nu} a)\overline{b}\langle a\rangle.!a(y).\overline{y}\langle t\rangle) :: d{:}1}\,\text{cut!}}{\Gamma;\cdot \vdash (\boldsymbol{\nu} z)((!z(m).m(u).\mathbf{0}) \mid (\boldsymbol{\nu} b)((!b(x).\overline{x}\langle z\rangle.(\mathbf{0} \mid \mathbf{0})) \mid (\boldsymbol{\nu} a)\overline{b}\langle a\rangle.!a(y).\overline{y}\langle t\rangle)) :: d{:}1}\,\text{cut!}$$

$*^1$: where, $A = (1{\multimap}1)$

REMARKS:

(i) In $*^1$, the next session depends on the type of channel $m$. Since $m$ receives input $u$ and then both of them stop showing any behavior, we can state that the type of this behaviour is $1{\multimap}1$.

$$m{:}(1{\multimap}1) \;\text{ and }\; z{:}!(1{\multimap}1).$$

(ii) Let us consider the session in branch $*^2$:

$$\Gamma, z{:}A;\cdot \vdash \overline{x}\langle z\rangle.(\mathbf{0} \mid \mathbf{0}).$$

This session describes the part of the process where server channel $z$ gets activated by server channel $a$ by passing $z$ along $a$.[2] Therefore, it is important for the $z$ in this session to be the same $z$ that is a server channel, that is the $z$ from the first sub-tree, which is well-typed. However, in CPT any form of output is bound, that is, a new channel is always created before it gets sent along another channel [Caires et al., 2012]. Therefore, it is not possible to type this session in CPT because there are no typing rules that allow for an existing channel ($z$) to be output along another channel ($x$). A correct session that would be well-typed in CPT is:

$$\Gamma, z{:}A;\cdot \vdash (\boldsymbol{\nu} z)\overline{x}\langle z\rangle.(\mathbf{0} \mid \mathbf{0}),$$

however, this process describes a different behaviour from that of $P_2'$. Finally, the judgment over $z$ that exists, is part of the shared context. The *copy* rule can be applied to a channel in the shared context that is used to activate a server, however, that is not the case in this session. Given our reasoning, it is not possible to type this session in CPT.

(iii) The session in branch $*^3$ allows for the application of the *copy* rule. However, given the contradiction in (ii), we can already conclude that it is not possible to construct a complete derivation tree for $P_2'$ in CPT.

[2]From the reduction steps mentioned before, server $b$ first activates server $a$, which then activates server $z$.

***Conclusion***
Following remarks [(i)-(iii)], since a complete derivation tree for $P_2'$ in CPT cannot be built, we can state that our assumption that $P_2'$ is well-typed in CPT is false. Therefore, $P_2$ is only well-typed in DS. We may conclude that 6.2 holds. □

$P_2$ represented a process where three servers were running in parallel. Every server was responsible for initiating another one. As such, the servers were bound by one another, even though they were not in the same process. This showcases how restrictive the typing rules of CPT are, where server channels cannot be bound by another channel, neither explicitly nor implicitly. In this thesis, we consider a server to be bound explicitly when its type depends on another channel. For example, in $a(x).!x(y)$, server $x$ is bound explicitly by channel $a$. We consider a server to be bound implicitly when its type does not depend explicitly on another channel, however its activation does depend on another channel. For example, in $!b(x).\overline{x}\langle z\rangle \mid \overline{b}\langle a\rangle.!a(y).\overline{y}\langle t\rangle$[3], server channel $a$, depends on server channel $b$ for activation. We now present the third process, which also showcases a non-static behaviour, but with two servers running concurrently.

### 6.3. **Third process.**

**Lemma 6.3.** $P_3$ is well-typed in DS, but not in CPT.

*Proof.* Similar to the other two lemmas, we will first show why $P_3$ is well-typed in DS by building its complete derivation tree using the typing rules in §5.2, then introduce what we assume to be the analogous process of $P_3$ in the syntax of CPT. Finally, we will construct the derivation tree for $P_3'$ in CPT and discuss the conclusions. $P_3$ is a parallel composition of four sub-processes:

$$P_3 \stackrel{\text{def}}{=} !a(y).z(t) \mid !b(x).x(y) \mid \overline{a}y \mid \overline{b}x.$$

It is similar to $P_2$ in that it is non-static and both servers get activated. However, it differs from $P_2$ in that the servers are not initiated by one-another, but rather, run independently of one another.

6.3.1. *Typability in DS.* We begin by constructing the complete derivation tree of $P_3$ in DS:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{z : \#M \;\; t : M \vdash \mathbf{0}}\text{T-nil}}{a : \#^n T'' \;\; y : T'' \;\vdash z(t)}\text{T-in}}{\vdash\; !a(y).z(t)}\text{T-rep} \qquad \dfrac{\dfrac{\dfrac{}{x : \#N \;\; y : N \vdash \mathbf{0}}\text{T-nil}}{b : \#^m T''' \;\; x : T''' \;\vdash \overline{x}y \quad lv(x) < m}\text{T-out}}{\vdash\; !b(x).\overline{x}y}\text{T-rep}}{\vdash\; !a(y).z(t) \mid !b(x).\overline{x}y}\text{T-par} \qquad \dfrac{\dfrac{}{a : \#T' \;\; y : T' \vdash \mathbf{0}}\text{T-nil}}{\vdash \overline{a}y}\text{T-out}}{\vdash\; !a(y).z(t) \mid !b(x).\overline{x}y \mid \overline{a}y}\text{T-par} \qquad \dfrac{\dfrac{}{b : \#T \;\; n : T \vdash \mathbf{0}}\text{T-nil}}{\vdash\; \overline{b}n}\text{T-out}$$

$$\vdash\; !a(y).z(t) \mid !b(x).\overline{x}y \mid \overline{a}y \mid \overline{b}n \qquad \text{T-par}$$

NOTE: For styling reasons, we assume that every channel type is carried on from each lower branch to an upper one.

REMARKS:
Since every leaf of the tree is an axiom, then $P_3$ is well-typed in DS, given the following constraints on the link type channels:

$$a : \#^n T'' \;\; b : \#^m T''' \;\; z : \#M \;\; x : \#N \quad \text{and} \quad lv(x) < m.$$

Thus, as long as the level of channel $x$ is strictly smaller than the level of channel $b$, then $P_3$ is well-typed in DS. As a consequence, this process is strongly normalizing.

[3]Note that in this brief example, we do not consider the correct syntax of CPT, for instance whether restriction is required or not.

6.3.2. *Typability in CPT.* We now consider a process $P_3'$ to be the analogous process of $P_3$ in CPT. Once again, we assume that $P_3'$ is well-typed in CPT, so that we can use the typing rules from §4.4 to construct the derivation tree. Let us first consider the structure of $P_3'$ in the syntax of CPT:

$$P_3' \stackrel{\text{def}}{=} (\boldsymbol{\nu} b)((!b(x).(\boldsymbol{\nu} y)\overline{x}\langle y\rangle.(\mathbf{0} \mid \mathbf{0})) \mid (\boldsymbol{\nu} a)((!a(y).z(t)) \mid (\boldsymbol{\nu} d)\overline{a}\langle y\rangle.(\mathbf{0} \mid \mathbf{0}) \mid \overline{b}(x).(\mathbf{0} \mid \mathbf{0})))$$

We note that the following:

(i) The channels that are responsible for the behavior of the process are $a, z, b$ and $x$.

(ii) Since $z$ is a free channel in $P_3'$, we can create judgement $z : C$, with $C$ being an arbitrary type that we can determine further.

(iii) In order for $P_3'$ to showcase the same behaviour as $P_3$, we need to represent two processes running in parallel with one another. In CPT, parallel behaviour can be typed using either the *cut* rule or the *cut!* rule [Caires et al., 2012]. Channels $a$ and $b$ can be used to apply the *cut!* rule when they are used as servers and the *cut* rule when they are used to activate the servers.

(iv) The application of the *cut!* rule

- We will first have to apply the *cut!* rule on server $b$. This will create two new branches:
  - One branch for the process that follows the server: $(\boldsymbol{\nu} y)\overline{x}\langle y\rangle.(\mathbf{0} \mid \mathbf{0})$. The succedent of this process will be the judgement over the server's input $x$.
  - The other branch is for the process that runs in parallel with the server: $(\boldsymbol{\nu} a)((!a(y).z(t)) \mid (\boldsymbol{\nu} d)\overline{a}\langle y\rangle.(\mathbf{0} \mid \mathbf{0}) \mid \overline{b}(x).(\mathbf{0} \mid \mathbf{0}))$. The succedent of this process is the judgment over $z$.
- Then we will also have to apply the *cut!* rule for server $a$. This will also create two branches:
  - One branch is for the process that follows server $a$: $z(t)$. The succedent of this process will be the judgement over $y$.
  - The other branch is for the process that runs in parallel with server $a$: $(\boldsymbol{\nu} d)\overline{a}\langle y\rangle.(\mathbf{0} \mid \mathbf{0}) \mid \overline{b}(x).(\mathbf{0} \mid \mathbf{0})$. The succedent of this process is the judgment over $z$.

(v) We choose the judgment over $z$ to be the succedent of this process; the linear context is empty. We assume that the shared context is $\Gamma$.

Given our reasoning [(i)-(v)], we present the derivation tree for $P_3'$ in CPT.

$$\dfrac{\dfrac{\dfrac{}{\Gamma;\cdot \vdash \mathbf{0} :: x{:}1}\,1\text{R} \quad \dfrac{}{\Gamma;\cdot \vdash \mathbf{0} :: y{:}1}\,1\text{R}}{\Gamma;\cdot \vdash (\boldsymbol{\nu} y)\overline{x}\langle y\rangle.(\mathbf{0} \mid \mathbf{0}) :: x{:}B}\,\otimes\text{R},\, *^1 \quad \dfrac{\Gamma, b{:}B, a{:}A;\cdot \vdash z(t) :: y{:}A^{*^2} \qquad \Gamma, b{:}B, a{:}A;\cdot \vdash (\boldsymbol{\nu} d)(\overline{a}\langle y\rangle.(\mathbf{0} \mid \mathbf{0}) \mid \overline{b}\langle x\rangle.(\mathbf{0} \mid \mathbf{0})) :: z : C^{*^3}}{\Gamma, b{:}B;\cdot \vdash (\boldsymbol{\nu} a)(!a(y).z(t) \mid (\boldsymbol{\nu} d)(\overline{a}\langle y\rangle.(\mathbf{0} \mid \mathbf{0}) \mid \overline{b}\langle x\rangle.(\mathbf{0} \mid \mathbf{0}))) :: z : C}\,\text{cut!}}{\Gamma;\cdot \vdash (\boldsymbol{\nu} b)((!b(x).(\boldsymbol{\nu} y)\overline{x}\langle y\rangle.(\mathbf{0} \mid \mathbf{0})) \mid (\boldsymbol{\nu} a)((!a(y).z(t)) \mid (\boldsymbol{\nu} d)\overline{a}\langle y\rangle.(\mathbf{0} \mid \mathbf{0}) \mid \overline{b}(x).(\mathbf{0} \mid \mathbf{0}))) :: z{:}C}\,\text{cut!}$$

$*^1$: where, $B = (1 \otimes 1)$

REMARKS:

(i) In $*^1$, the application of the rule depends on the type of channel $x$. Since channel $x$ behaves as a bound output and the rest of the processes are inert, then we define B to be $1 \otimes 1$.

(ii) We note that the first sub-tree is complete since both its leaves are axioms. Therefore, the initial type of $x$ is $x{:}1 \otimes 1$.

(iii) We consider the current session from the sub-tree in $*^2$:

$$\Gamma;\cdot \ \vdash z(t) :: y{:}A$$

This session depends on the judgement over channel $z$, which is receiving input, but no judgment over $z$ is present. The application of the *cut!* rule resulted in this process obtaining shared channel $a$ in its shared context and judgment over $y$ as its succedent. With no judgment

over the channel of the current session present, it is not possible to apply any of the tying rules from §4.4. As a result, this session is the final form that we are able to obtain from this sub-tree, which is not an axiom.

(iv) If we consider the sub-tree in $*^3$, we note that it is possible to apply the $cut$ rule and obtain two new branches for $\overline{a}\langle y\rangle$ and $\overline{b}\langle x\rangle$. However, from (iii), since we have concluded that one of the sub-trees does not result in an axiom, we do not continue with the construction of the rest of the derivation tree.

***Conclusion***

Since it is not possible to construct a complete derivation tree for $P_3'$ in CPT, then $P_3'$ is not well-typed in CPT. Therefore, our assumption was false. This confirms 6.3. □

Non-static process $P_3$ could not be typed in CPT despite the servers running independently from one another, because the necessary judgment to type one of the sub-processes was missing. Despite defining the judgment in the root of the derivation tree, the distribution rendered by the typing rules did not allow for that judgment to be placed in the sub-process that would acquire it later on. More specifically, the *cut!* rule allows for two sub-processes to separate, where the sub-process that follows the server receives no linear context. Therefore, for this sub-process to be typed in CPT, it is necessary that every channel that follows the server be bound by it, so that no antecedents will be needed for its typing. We will now present a fourth and final process that is well-typed in DS and CPT. Similar to the previous processes, we will first introduce the complete derivation tree of the process in DS, then the analogous process in CPT, along with its complete derivation tree in CPT.

## 6.4. **Fourth process.**

**Lemma 6.4.** $P_4$ is well-typed in DS and CPT.

*Proof.* We will now consider a fourth process $P_4$ that is well-typed in both type systems. $P_4$ is a non-static process made up of two sub-processes running concurrently, where one of them represents a server and the other one, the client process that activates the server. Given the process:

$$P_4 \stackrel{\text{def}}{=} \ \overline{a}x.x(y) \mid !a(b).\overline{b}c,$$

we note that the client process on the left activates the server $a$ on the right by passing along $a$ channel $b$. The server uses channel $b$ to output some message $c$. The client, on the other hand, upon sending the request, receives another message along its channel $x$ and halts. We begin by constructing the derivation tree of $P_4$ in DS, then we introduce what we assume to be the analogous of $P_4$ in CPT and construct its derivation tree.

### 6.4.1. *Typability in DS.*

We can build the derivation tree for $P_4$ using the typing rules from §5.2.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{b : \#T' \ \ c : T' \ \ \vdash \mathbf{0}}\text{T-nil}}{a : \#^n T \ \ b : T \ \ \vdash \overline{b}c \ \ \ lv(b) < n}\text{T-out}}{\vdash !a(b).\overline{b}c}\text{T-rep} \qquad \dfrac{\dfrac{\dfrac{}{x : \#T'' \ \ y : T'' \ \ \vdash \mathbf{0}}\text{T-nil}}{a : \#^n T \ \ x : T \ \ \vdash x(y)}\text{T-in}}{\vdash \overline{a}x.x(y)}\text{T-out}}{\vdash \overline{a}x.x(y) \mid !a(b).\overline{b}c}\text{T-par}$$

NOTE: For styling reasons, we assume that every channel type is carried on from each lower branch to an upper one.

REMARKS:
The derivation tree is complete as both its leaves are axioms. This process is well-typed in DS if and only if the following typing of the link type channels is preserved:

$$a : \#^n T \;\; b : \#T' \;\; x : \#T'' \text{ and } lv(b) < n$$

Therefore, as long as the level of channel $b$ is strictly smaller than the level of server channel $a$, then $P_4$ is well-typed in DS. Since, $b$ will be substituted for $x$ when the server gets activated, then the level of channel $x$ must be strictly smaller than the level of server $a$: $lv(x) < n$. Since $P_4$ is well-typed in DS, we can conclude that $P_4$ is strongly normalizing in DS.

6.4.2. *Typability in CPT.* We now consider the same process according to the syntax of CPT by assuming that $P_4'$ is analogous to $P_4$ in CPT. If we assume that $P_4'$ is well-typed in CPT, then we are able to construct its derivation tree using the typing rules from §4.4, where:

$$P_4' \stackrel{\text{def}}{=} \; (\boldsymbol{\nu} a)((\boldsymbol{\nu} x)\overline{a}\langle x\rangle.x(y).\mathbf{0} \mid !a(b).(\boldsymbol{\nu} c)\overline{b}\langle c\rangle.(\mathbf{0} \mid \mathbf{0}))$$

We note the following:

(i) The channels that are responsible for the behaviour of the process are $a, x$ and $b$.

(ii) This process is made up of two sub-processes that run in parallel, both of which have channel $a$ in common. As a result, using the *cut* rule, channel $a$ can be used as communicating link between the two.

(iii) There are no free channels in this process. Channels $x$ and $b$ are both bound by $a$. Channel $a$ is restricted to both sub-processes and no judgement can be created for its type. Therefore, the succedent of this process will be a dummy variable $d$:$\mathbf{1}$; the linear context will be empty. We assume the shared context to be $\Gamma$.

(iv) The application of the *cut* rule.
Applying the *cut* rule to $P_4'$ leads to the creation of two separate processes, one which offers a service via channel $a$, and one which makes use of the same service via channel $a$ (§4.4). Since $P_4'$ is a case of client-server communication, we know that the client side will have to make use of the *copy* rule, whereas the server side will make use of the *!R* rule [Caires et al., 2012]. Thus, we reason that channel $a$ provides the service of a replicated input: $a$:!$A$, where:
- the server sub-process offers it: $\vdash !a(b).(\boldsymbol{\nu} c)\overline{b}\langle c\rangle.(\mathbf{0} \mid \mathbf{0}) :: a{:}!A$
- the client sub-process uses it: $a{:}!A \vdash \overline{a}\langle x\rangle.x(y).\mathbf{0}$. That is, $a$:!$A$ will be part of its linear context, when the *cut* rule is applied.[4]

Given the insights [(i)-(iv)], we now construct the derivation tree for $P_4'$.

[4]Note that for styling matters, we have not used the complete syntax to showcase how the application of the *cut* rule affects the separation of the two sub-processes.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, u{:}A; \cdot \vdash \mathbf{0} :: d{:}\mathbf{1}}\,1\mathrm{R}}{\Gamma, u{:}A; x{:}\mathbf{1} \vdash \mathbf{0} :: d{:}\mathbf{1}}\,1\mathrm{L}}{\Gamma, u{:}A; x{:}\mathbf{1}, y{:}\mathbf{1} \vdash \mathbf{0} :: d{:}\mathbf{1}}\,1\mathrm{L}}{\Gamma, u{:}A; x{:}A \vdash x(y).\mathbf{0} :: d{:}\mathbf{1}}\,\otimes\mathrm{L},{*}^1}{\Gamma, u{:}A; \cdot \vdash (\boldsymbol{\nu} x)\overline{u}\langle x\rangle.x(y).\mathbf{0} :: d{:}\mathbf{1}}\,\mathrm{copy}}{\Gamma; a{:}!A \vdash (\boldsymbol{\nu} x)\overline{a}\langle x\rangle.x(y).\mathbf{0} :: d{:}\mathbf{1}}\,!\mathrm{L} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma; \cdot \vdash \mathbf{0} :: b{:}\mathbf{1}}\,1\mathrm{R} \quad \dfrac{}{\Gamma; \cdot \vdash \mathbf{0} :: c{:}\mathbf{1}}\,1\mathrm{R}}{\Gamma; \cdot \vdash (\boldsymbol{\nu} c)\overline{b}\langle c\rangle.(\mathbf{0} \mid \mathbf{0}) :: b{:}A}\,\otimes\mathrm{R},{*}^2}{\Gamma; \cdot \vdash !a(b).(\boldsymbol{\nu} c)\overline{b}\langle c\rangle.(\mathbf{0} \mid \mathbf{0}) :: a{:}!A}\,!\mathrm{R}$$

$$\dfrac{\Gamma; a{:}!A \vdash (\boldsymbol{\nu} x)\overline{a}\langle x\rangle.x(y).\mathbf{0} :: d{:}\mathbf{1} \qquad \Gamma; \cdot \vdash !a(b).(\boldsymbol{\nu} c)\overline{b}\langle c\rangle.(\mathbf{0} \mid \mathbf{0}) :: a{:}!A}{\Gamma; \cdot \vdash (\boldsymbol{\nu} a)((\boldsymbol{\nu} x)\overline{a}\langle x\rangle.x(y).\mathbf{0} \mid !a(b).(\boldsymbol{\nu} c)\overline{b}\langle c\rangle.(\mathbf{0} \mid \mathbf{0})) :: d{:}1}\,\mathrm{cut}$$

${*}^1, {*}^2$: where, $A = 1 \otimes 1$

REMARKS:

(i) The derivation tree is complete in CPT because all three leaves are axioms, that is, they have resulted from applying the rule $1R$.

(ii) In ${*}^1$, we need to apply the rule that concerns the type of channel $x$. Since channel $x$ input $y$ and then the process terminates, we can discern that A represents the type $1 \otimes 1$, because channel $x$ is an antecedent. For this reason, the left rule of the operation $\otimes$ is applied.

(iii) In ${*}^2$, the type of $A$ has already been calculated to be $1 \otimes 1$, except this time, it pertains to channel $b$, which is a succedent. Therefore, the right rule of the operation $\otimes$ is applied.

Given the remarks, we can conclude that $P_4'$ is well-typed in CPT. Since, $P_4'$ is well-typed in CPT, then $P_4'$ is strongly normalizing.

***Conclusion***

By constructing the complete derivation trees for the fourth process in both type systems, we can conclude that $P_4$ is well-typed in DS and CPT (provided the analogous process in CPT). Therefore, $P_4$ is strongly normalizing according to the typing rules of both, DS and CPT. Finally, we are able to confirm 6.4. □

With this fourth and final conclusion, we have reached the end of our analysis for the comparison between DS and CPT. We considered four processes that were different from one another, which resulted in four Lemmas: 6.1, 6.2, 6.3 and 6.4. The first three lemmas stated that processes $P_1$, $P_2$ and $P_3$ respectively, are well-typed in DS, but not in CPT. The fourth lemma introduced a fourth non-static process that is well-typed in both type systems. We proved the lemmas by constructing the derivation trees for each process using the typing rules from §4.4 and §5.2. We will now present our final conclusions of this thesis by introducing our insights from the research and the processes.

## 7. Conclusions

Having studied two different type systems for concurrent processes, we are now able to state some conclusions of our own. First of all, it is important to note that both type systems seek to prove the termination property by means of carefully curated rules that can establish whether a process belongs to a certain family of processes or not. The idea is fairly simple: should a process be well-typed in a type system, then that process is strongly normalizing, that is, it terminates. The initial point of interest in this research was to study both type systems in the hopes of determining a group of processes that are well-typed in both of them. The study of concurrent processes poses great interest in the field of software and not only, and as such, attracts many researchers. Every researcher has their own outlook on the matter and until a standard method can be defined for concurrent processes, it will be the case that several type systems are available. For this reason,

comparing the most common ones poses an interesting and separate field of research as well. The most interesting aspect to consider is how each type system handles the behaviour of a server channel. The result of our current work is the study of a subset of rules from each type system, and how they are able to define the channel types and the evolution of a process. Using solely this subset of rules, we were able to discern three different processes that are well-typed in one type system, but not in the other. What is more, the rules that we consider are analogous to one another; that is, if we consider how an input channel is typed in DS, then we do the same for CPT.

7.0.1. *What is common for the two?* Both type systems are based on rules, called typing rules, that allow us to discern whether a process can be well-typed in the type system in question or not. Furthermore, both of them pose restrictions on the definition of the rule about a replicated server, however, their approaches are quite different. DS and CPT both rely on $\pi$-calculus to structure their processes and typing rules. They share a very similar syntax with one another, except for a few minor differences with the output prefix and the restriction process. The rules of both typing systems can be used to build derivation trees that allow us to conclude whether a process is well-typed or not.

7.0.2. *Differences:* We will now discuss the differences between DS and CPT by focusing on certain concepts and insights from our research.

(i) CHANNEL TYPES:
In CPT, a type is used to describe the behaviour of a channel. In doing so, we are able to describe the behaviour of the complete process. Types in CPT are all based on logical connectives. It is not possible to apply a typing rule without knowing the channel type of the current session.
On the other hand, in DS, a type is used to discern whether a channel can pass along other channels, in which case it is a link type channel, or simply pass along *numbers* and *booleans*, in which case it is a basic type channel. Link types are denoted with the pound symbol #. The typing rules in DS help us conclude the final types of every channel.

(ii) LINEARITY OF CHANNELS:
CPT is based on linear logic and as such, is bound to be more restrictive than DS. When a channel performs according to its type, it changes its state and cannot perform with the same type again. Instead, it relies on the new type, a notion also known as *session types*.
In DS, there are no session types. Channels have one single type throughout the entirety of the process. This type encompasses every behaviour that is required of the channel and can only be defined with certainty when the derivation tree is complete.

(iii) LOGIC CONCEPTS:
CPT borrows other concepts from logic as well, including antecedents, succedent, and judgments. Antecedents and succedents stand for channels but depend on the position of the channel in the process, which will be either to the left or to the right, respectively. However,in DS, antecedents or succedents do not have a role as impactful as in CPT.

(iv) LEVELS:
In DS, we are introduced with the concept of *levels* for channels. Every link type channel has a level, and the point of levels is to make sure that no circular motion can be initiated between two (or more) servers that are running in parallel. Therefore, levels are crucial to preserving the termination property for processes. Levels do not exist in CPT.

(v) JUDGMENTS:
A judgment in CPT states the *type* of a channel. Moreover, the set of judgments to the left of a process form the *context* of a process, which is an essential tool for determining the typability of a process in CPT. There are two types of contexts: linear and shared. The channels that never change state belong to the shared context, and the other channels belong to the linear context. It is important to note that the typing rules of CPT allow for the typing of linear channels only. The *copy* rule, being the only exception, merely upgrades the status of a channel from linear to shared, without any further changes.
In DS, a judgment is merely used to state whether a *process* is well-typed in it, or not.

(vi) DERIVATION TREES:
In CPT a derivation tree is complete if and only if the leaves are axioms. If rule ***1****R* cannot be applied on every sub-tree, then the tree is not complete. Therefore, it is not possible to always apply a particular typing rule. This is because rules are tightly connected to the types of the channels. If a channel is present in the process but is missing a specific type, then it is very likely that no typing rules can be applied to the process.
On the other hand, in DS, it is always possible to apply one of the typing rules, because there are no restrictions placed by the types of the channels; we only consider the current behaviour of the channel. Whether a process is well-typed in DS is determined by the constraints placed by the typing rules once the derivation tree is complete.

7.0.3. *Insights from $P_1$, $P_2$, $P_3$.* In this paper, we have shown that it is possible to construct the derivation trees in DS for three different processes: a static process, and two non-static ones, but it is not possible to do so in CPT. The later one is very restrictive with regard to the behaviour of the replicated input. Given the sub-processes that were well-typed in CPT, it appears that in a process with a replicated server, every channel that follows the server, should be bound by the server, or the server's input and so on, in a pipeline manner. Therefore, as long as no antecedents are present when we are considering a session with a server, the process will be well-typed in CPT. We now present the following conclusions from the three processes with regard to CPT:

(1) In the first process, we were unable to complete the derivation tree because we used the left rule of the *bang* operation, which merely substitutes the server channel with a channel of the shared context, instead of the linear one. This situation occurs when the judgment over the server channel is an antecedent. Since the server was firstly received along channel $a$, the relevant input typing rule rendered it bound by that channel. As a result, it was not feasible to construct a complete derivation tree for $P_1$.
**Conclusion**: A server channel in CPT should not be an antecedent and/or bound by another channel otherwise, the process is not well-typed. Therefore, explicit boundness is not allowed in CPT.

(2) In the second process, we were unable to construct a complete derivation tree, because the three server channels were initiated by one another. This situation constrained us to use only the *cut!* rule, however, the division of the contexts and judgments, made it impossible for the process to be typable in CPT.
**Conclusion**: Two or more servers cannot initiate communication with one another in CPT otherwise, the process is not well-typed in CPT. Therefore, implicit boundness is not allowed in CPT.

(3) The third process was different from the second one because servers did not initiate communication with one another, however, the fact that they were both activated, resulted in a situation

similar to $P_2$. The division of contexts and judgments due to the *cut!* rule made it impossible for us to construct a full derivation tree in CPT.
**Conclusion**: Two independent servers should not be initiated in the same process in CPT otherwise, the process in question is not well-typed in CPT.

7.0.4. *Conclusion:* To summarise, a multitude of processes can be typed in DS because the typing rules allow for channels to showcase different types of behaviours in the same process. As such, it is possible to type processes with several servers in it, static or non-static, as long as the types of channels abide by the level constraints. In CPT, we have identified a few situations which cannot be well-typed due to the restrictions of the typing rules. In this thesis, we introduced two concurrency type systems and performed a comparative analysis of the two, by focusing on four processes. We introduced $\pi$-calculus as the programming language for the processes. We also presented the typing rules for both type systems and showcased how processes can be typed in them. Afterwards, we conducted an extensive analysis of the four processes. In the end, we summarized the points of commonality and differentiation while also introducing our final thoughts and insights.

## References

[B. MacQueen, 2012] B. MacQueen, D. (2012). Quick introduction to type systems.
[Braüner, 1996] Braüner, T. (1996). *Introduction to Linear Logic*.
[C. Pierce, 1995] C. Pierce, B. (1995). Foundational calculi for programming languages. *CRC Handbook of Computer Science and Engineering*.
[Caires et al., 2012] Caires, L., Pfenning, F., and Toninho, B. (2012). Towards concurrent type theory. *Invited talk at TLDI'12*.
[Cardelli, 2004] Cardelli, L. (2004). Type systems. *CRC Handbook of Computer Science and Engineering (2nd ed.)*.
[Deng and Sangiorgi, 2006] Deng, Y. and Sangiorgi, D. (2006). Ensuring termination by typability. *Information and Computation 204 (2006) 1045-1082*.
[Lan, 2019] Lan, H. (2019). A crash course on proving the halting problem.
[Milner et al., 1992] Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes pt.1. *Information and Computation 100(1) pp.1-40*.
[Parrow, 2001] Parrow, J. (2001). An introduction to the $\pi$-calculus.
[Wadler, 2014] Wadler, P. (2014). Propositions as sessions. *Journal of Functional Programming, 24(2-3), 384-418*.
[Wadler, 2015] Wadler, P. (2015). Propositions as types. *Communications of the ACM*.