DEPARTMENT OF COMPUTER SCIENCE

Concurrency with Classical Linear Logic

Elizabeth Sidebottom

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Thursday 5th May, 2022

Abstract

We introduce classical linear logic, a substructural logic with a classical framework. Then we explore classical processes, a prototypical concurrent language which presents classical linear logic as a session typed process calculus. We then provide a translation from the simply typed lambda calculus into classical processes with a proof of simulation. This can provide the logical foundation for an enhanced message passing concurrent programming language which makes use of session types to provide type checking.

Dedication and Acknowledgements

I dedicate this project to Willow who saw me start this project but wasn't here to see it complete, and to Inca who has been with me every step of the way.
I would also like to thank both my supervisor, Dr. Alex Kavvos for his invaluble help with this project, and my boyfriend, Cam who is always supportive of everything I do.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted or otherwise incorporated material which is the work of others, I have included the source in the references. Any views expressed in the dissertation, other than referenced material, are those of the author.

Elizabeth Sidebottom, Thursday 5th May, 2022

Contents

List of Figures

Ethics Statement

This project did not require ethical review, as determined by my supervisor, Dr. Alex Kavvos.

Chapter 1

Introduction

There are two forms of logic: classical logic, and constructive logic (also called intuitionistic logic). Classical logic is the foundation for classical mathematics and has a few defining features including logical operations, structural rules, and double negation. Logical operations connect two or more statements such as $+, \otimes$, etc. Structural rules are operations performed directly on judgements or sequents such as weakening and exchange. Double negation is the assertation that if a proposition P is true then the inverse of it's inverse is also true, $P = \neg\neg P$. Statements in classical logic can be either true or false, and cannot be undetermined. More specifically, if we have some proposition P, either P is true, or the negation of P is true. This means we can use proof by contradiction to show that a proposition is true if its negation is false. Put simply, in classical logic, a statement is true if it is not false.

In constructive logic, we have a stronger notion of truth. We only accept a proposition as true if it has a constructive proof, so in order to prove an object exists we must either provide an example of such an object or explain how we would create it. Dually, we only accept a proposition to be false if we have a refutation of it in constructive logic. There is no expectation that a proposition be either true or false, it can simply be a proposition. Constructive logic is a logic of positive evidence which makes it more expressive than classical logic. Any proof of a proposition in classical logic can be translated into a proof of a weaker (although calssically equivalent) proposition in constructive logic.

Our main topic is classical linear logic (CLL) which was first presented by Girard [5] as an extension of constructive logic with a classical framework. CLL is a substructural logic which makes use of double negation from classical logic, and constructivism from intuitionistic logic. The addition of exponentials allows us to use structural rules to some extent. It has a number of applications in computer science, namely in reasoning about resources, resource useage, ownership, and communication. The latter is seen under the Curry-Howard isomorphism and is the aspect we will be focusing on.

The Curry-Howard isomorphism is a correspondence between proof systems and type systems where we see propositions as types, proofs as programs, and normalisation as computation. Here we use a variant of this found by Caires and Pfenning [2] with

propositions as session types,
proofs as processes,
cut elimination as computation.

This variant equates classical linear logic with a process calculus similar to the π-calculus and allows us to model classical linear logic as a parallel programming language. For this report we will use Wadler's [10] variant on the π-calculus which he combines with CLL to create a session-typed process calculus named classical processes (CP). Wadler's variant extends the π-calculus so it has processes corresponding to each of the typing judgements needed for CLL. For each logical proposition we have a session type which describes a type for the communicating protocols. Session types allow us to describe how two processes communicate along a channel. They are most often used in relation to the π-calculus which was introduced by Milner [8] as a way to model processes which communicate concurrently.

There is no lack of literature on classical linear logic, however almost all of it assumes a reasonable level of prior knowledge on the subject. This would not be a problem if someone had written down all the rules

and explained the purpose of CLL. Unfortunately, no one thought to do such a thing and so the rules are taught to students by supervisors in a one-on-one setting. The first aim of this thesis is to present the rules of classical linear logic such that someone who has no prior knowledge can come away knowing enough to approach more literature on the topic without feeling out of depth. We will then provide a translation from the simply typed lambda calculus into classical linear logic which has been considered by many prominent figures, but never been written out in full.

1.1 Outline

We begin with chapter 2 where we discuss the simply typed lambda calculus chapter, detailing the statics and dynamics so we have a good foundation for our translation. We then introduce classical linear logic in chapter 3, explaining the original meanings of the propositions, and providing an intuition for how they combine with session types to create classical processes. We follow a similar trajectory to the previous chapter, outlining both the statics and dynamics for CP so it is easy for the reader to make a direct comparison between the two calculi. In chapter 4 we give a call-by-name translation from the λ-calculus into CP and prove substitution for the translation. We also provide a proof of simulation to show that our translation is correct.

Chapter 2

The Simply Typed Lambda Calculus

The lambda calculus was invented by Church the 1930s [3] to provide a logical foundation for mathematics. However, his original system was soon proved to have some logical inconsistencies [6]. Church then published the section of his system meant for dealing with functions in isolation which he proved to be consistent. This portion of his original work is now known as the untyped lambda calculus. Later on, he published the simply typed lambda calculus [4] which is also logically consistent although computationally weaker than its untyped counterpart. Although his original system was designed as a foundation for mathematics, it was largely overlooked by the mathematic community who preferred set theory. It wasn't until the 1960s that the λ-calculus was discovered to be a useful tool in computer science.

The simply typed lambda calculus (λ-calculus) is regarded as the most basic functional programming language. It can be used both as a simple programming language with which we can execute computation, and also as a mathematical object which we can perform proofs upon. Both of these aspects are extremely useful to computer scientists. Modelling computation helps us to understand how a language works, and viewing a language as a mathematical object allows us to prove that it is safe.

2.1 Statics

Types	τ	::=	1	unit
			$\tau_1 + \tau_2$	sum
			$\tau_1 \times \tau_2$	product
			$\tau_1 \to \tau_2$	function
Terms	e	::=	x	variable
			$\mathsf{inl}(e)$	left injection
			$\mathsf{inr}(e)$	right injection
			$\mathsf{case}(e; x.\, e_1; y.\, e_2)$	case
			$\langle e_1, e_2 \rangle$	pair
			$\pi_1(e)$	left projection
			$\pi_2(e)$	right projection
			$\lambda x : \tau.\, e$	abstraction
			$e_1(e_2)$	application

Figure 2.1: Types and Terms for the λ-Calculus

The simply typed lambda calculus is a language for working with functions. If we have some typing relation $\Gamma \vdash x : \sigma$ this means that in the context Γ, x has type σ, and x is said to be well-typed. We can see from 2.1 that we have three composite types, and terms can take on specific forms, the rules for which

are shown in 2.2. The sum type is used to represent disjoint union so if $\Gamma \vdash x : \tau_1 + \tau_2$ then the type of x is either τ_1 or τ_2. The product type is used to represent pairs, so if we have $\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2$ then we know that we have both $e_1 : \tau_1$ and $e_2 : \tau_2$. Finally, the function type is of course used to represent functions so if we have $\lambda x : \sigma. e : \sigma \to \tau$ then this is equivalent to having some function $f : \sigma \to \tau$ so we must have that $x : \sigma$ and $f(x) = e : \tau$. So our function f takes in some parameter of type σ and outputs a result with type τ.

The rules in 2.2 have premises on the top and conclusions on the bottom. So for any rule we know that if we have everything above the line, then we can conclude anything below the line. Rules without any premises are called axioms and require no justification. We take a constructor to be a rule which can create something of a certain type, and a destructor to be a rule which reduces something of that type to a simpler type.

$$\textsc{Var}\ \frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash \langle\rangle : 1}$$

$$\textsc{In-l}\ \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{inl}(e) : \tau_1 + \tau_2} \qquad \textsc{In-r}\ \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{inr}(e) : \tau_1 + \tau_2}$$

$$\textsc{Case}\ \frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 \vdash e_1 : \tau \qquad \Gamma, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{case}(e; x.\, e_1; y.\, e_2) : \tau}$$

$$\textsc{Pair}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \qquad \textsc{Prj-l}\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1(e) : \tau_1} \qquad \textsc{Prj-r}\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2(e) : \tau_2}$$

$$\textsc{Abs}\ \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x : \sigma. e : \sigma \to \tau} \qquad \textsc{App}\ \frac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1(e_2) : \tau}$$

Figure 2.2: Static Rules for the λ-Calculus

The Var rule simply states that if we have $x : \sigma$ in some environment Γ, then we know x has type σ. The Unit rule states that if we have an empty pair, then this has the unit type, 1. In-r is a constructor for the sum type which allows us to use injection to derive that $\mathsf{inl}(e)$ has type $\tau_1 + \tau_2$ given that e has type τ_1, and similarly for In-l. The Case rule is the destructor for the sum type which branches on either side depending on the specific instance given. The Pair rule allows us to construct a product type $\tau_1 \times \tau_2$ given two terms with each of those types. Prj-l destructs some pair with a product type $\tau_1 \times \tau_2$ into a term with type τ_1, and similarly Prj-r will destruct the pair into a term with type τ_2. The Abs rule refers to lambda abstraction and is a constructor for the product type. It allows us to create a function $\lambda x : \sigma. e\sigma \to \tau$ if we can obtain some e of type τ from Γ and $x : \sigma$. The App rule is the destructor of the function type, function application. If we have some function $e_1 : \sigma \to \tau$ and some variable e_2 of type σ then we can apply the function e_1 to e_2 and obtain a result with type τ.

2.2 Dynamics

We know how the typing system works for the λ-calculus, but we also need to understand it's computational behaviour, i.e. the dynamics of a program. To understand these rules fully, we will need the substituition lemma.

Substituition: If we have $\Gamma \vdash e : \tau$ and $\Gamma, x : \tau \vdash u : \sigma$, then $\Gamma \vdash u[e/x] : \sigma$.

$$x[e/x] \stackrel{\text{def}}{=} e \qquad w[e/x] \stackrel{\text{def}}{=} w$$

$$\mathsf{inl}(u)[e/x] \stackrel{\text{def}}{=} \mathsf{inl}(u[e/x]) \qquad \mathsf{inr}(u)[e/x] \stackrel{\text{def}}{=} \mathsf{inr}(u[e/x])$$

$$\mathsf{case}(u; y.\, e_1; z.\, e_2)[e/x] \stackrel{\text{def}}{=} \mathsf{case}(u[e/x]; y.\, e_1[e/x]; z.\, e_2[e/x])$$

$$\langle\rangle[e/x] \stackrel{\text{def}}{=} \langle\rangle \qquad \langle e_1, e_2\rangle[e/x] \stackrel{\text{def}}{=} \langle e_1[e/x], e_2[e/x]\rangle$$

$$\pi_1(u)[e/x] \stackrel{\text{def}}{=} \pi_1(u[e/x]) \qquad \pi_2(u)[e/x] \stackrel{\text{def}}{=} \pi_2(u[e/x])$$

$$(\lambda w.\, u)[e/x] \stackrel{\text{def}}{=} \lambda w.\, u[e/x] \qquad (e_1(e_2))[e/x] \stackrel{\text{def}}{=} (e_1[e/x])(e_2[e/x])$$

Figure 2.3: Substitution for the λ-Calculus

Now we understand how substituition works, we can fully appreciate the dynamic rules for the λ-calculus which are shown below.

$$\textsc{Val-Unit}\ \frac{}{\langle\rangle\ \text{val}} \qquad \textsc{Val-In-L}\ \frac{}{\mathsf{inl}(e)\ \text{val}} \qquad \textsc{Val-In-R}\ \frac{}{\mathsf{inr}(e)\ \text{val}} \qquad \textsc{Val-Pair}\ \frac{}{\langle e_1, e_2\rangle\ \text{val}} \qquad \textsc{Val-Abs}\ \frac{}{\lambda x : \sigma.\, e\ \text{val}}$$

$$\textsc{D-Case-In-L}\ \frac{}{\mathsf{case}(\mathsf{inl}(e); x.\, e_1; y.\, e_2) \longmapsto e_1[e/x]} \qquad \textsc{D-Case-In-R}\ \frac{}{\mathsf{case}(\mathsf{inr}(e); x.\, e_1; y.\, e_2) \longmapsto e_2[e/y]}$$

$$\textsc{D-Case}\ \frac{e \longmapsto e'}{\mathsf{case}(e; x.\, e_1; y.\, e_2) \longmapsto \mathsf{case}(e'; x.\, e_1; y.\, e_2)}$$

$$\textsc{D-Prj-Pair-L}\ \frac{}{\pi_1(\langle e_1, e_2\rangle) \longmapsto e_1} \qquad \textsc{D-Prj-Pair-R}\ \frac{}{\pi_2(\langle e_1, e_2\rangle) \longmapsto e_2}$$

$$\textsc{D-Prj-L}\ \frac{e \longmapsto e'}{\pi_1(e) \longmapsto \pi_1(e')} \qquad \textsc{D-Prj-R}\ \frac{e \longmapsto e'}{\pi_2(e) \longmapsto \pi_2(e')}$$

$$\textsc{D-Beta}\ \frac{}{(\lambda x : \sigma.\, e_1)(e_2) \longmapsto e_1[e_2/x]} \qquad \textsc{D-App}\ \frac{e_1 \longmapsto e_1'}{e_1(e_2) \longmapsto e_1'(e_2)}$$

Figure 2.4: Dynamic Rules for the λ-Calculus

Again we have premises above the line and conclusions below the line. However, here we make use of $e_1 \longmapsto e_2$ to signify a transition from one term to another. This transition can be thought of as a change of state from e_1 to e_2. Clearly, any "val" rule simply states that terms of the given structure are values and thus cannot transition. Apart from these val rules, we have two distinct transition types: instruction transitions which perform computation and require no premises, and search transitions which allow for subterm computation. Together these determine the order of evaluation of terms. If we can perform a search transition on some term then we must do so before performing an instruction transition. For each instruction transition, there is an implication that none of the subterms can transition to another state while the whole term is in it's current state.

The D-Case-In-L rule substitutes e for x in e_1 if $\mathsf{inl}(e)$ is the first subterm, and D-Case-In-R performs the same substitution in e_2 if we have $\mathsf{inr}(e)$ instead. The D-Case rule transitions from a case statement with

e to the same case statement but with e' if e can transition to e'. D-Prj-Pair-l offers the left projection of a pair, and similarly D-Prj-Pair-r offers the right projection. D-Prj-l and D-Prj-r transition from the projection of e to the projection of e' if $e \longmapsto e'$. The D-Beta rule performs a beta reduction on a lambda term and replaces x with e_2 in e_1. Finally, D-App transitions from the application $e_1(e_2)$ to $e_1'(e_2)$ if $e_1 \longmapsto e_1'$.

Combining both the dynamics and statics, we can ensure that the λ-calculus is type safe if the following properties hold:

Progress: If we have some term e where $\vdash e : \tau$ either e is a value, or $e \longmapsto e'$ for some e'.

Preservation: If we have $\vdash e : \tau$ and $e \longmapsto e'$ we must have $\vdash e' : \tau$.

Progress states that computation will continue until evaluation is complete, i.e. until we have a value. Preservation tells us that types are preserved under evaluation. Together they show that well-typed programs don't go wrong.

Chapter 3

Classical Linear Logic and Classical Processes

Classical linear logic was introduced by Girard as the logic behind logic. It has a classical framework whilst also being constructive and was the first attempt at solving concurrency at a logical level. Here, we will build on the issue of concurrency by interpreting propositions as session types and extending CLL to CP, a session-typed process calculus introduced by Wadler. We will first explore CLL alone, then combine it with session types to produce a prototypical concurrent language.

In classical linear logic variables are denoted by capital letters $A, B, ...$ and the duals to these variables are denoted $A^{\perp}, B^{\perp}, ...$ where $A^{\perp}$ (not A) is the negation of A and $A^{\perp\perp} = A$. This negation is not present in intuitionistic linear logic since it retains the usual two sided sequent. However, in CLL we replace the two-sided sequent with a one-sided sequent, so a sequent in classical logic such as:

$$A_1, A_2, ..., A_n \vdash B_1, B_2, ..., B_m$$

$$\text{becomes} \quad \vdash {A_1}^{\perp}, {A_2}^{\perp}, ..., {A_n}^{\perp}, B_1, B_1, ..., B_m$$

This makes CLL slightly easier to read as we don't have to worry about propositions on the left-hand side of our sequent. We can see the use linear negation in our new one-sided sequent, as anything which would occur on the left-hand side of a classical sequent becomes a negation on the right-hand side of a CLL sequent. We also have that any proposition must be used exactly as many times as it appears. We cannot have some proposition A and not make use of it, and similarly we cannot use it any more than once if it only appears once.

3.1 The Vending Machine

Generally when researching the basics of classical linear logic, one may find the vending machine example which is widely used to describe the propositions of CLL in a way that is easy to understand. Say we have a vending machine which takes £1 coins and has the options of Tea or Coffee. If we see the tensor symbol on the machine: *Tea* $\otimes$ *Coffee*, then inserting £1 will give us both Tea and Coffee. However if we see *Tea* $\&$ *Coffee* then inserting £1 will give us a choice of either Tea or Coffee. If we can't decide what drink to have, then finding a vending machine which says *Tea* $\oplus$ *Coffee* will take our £1 and give us either Tea or Coffee making the decision for us. Par is not quite so easy to describe, if we stay with our vending machines then we would most likely see £1 $\parr$ *Tea* which would imply that inserting £1 into the vending machine will then give us Tea. Having a term $A \parr B$ in classical linear logic is equivalent to having either $A^{\perp} \multimap B$ or $B^{\perp} \multimap A$ in linear logic. Essentially, we can have either £1 or Tea, but having one means we have given up the other.

Moving on to the exponential fragment, if we have a term !£1 this would indicate that we have an unspecified number of £1 coins which we may use at the vending machine. We may also see a term ?£1 on any of these vending machines which would indicate that it is possible to use the machine any number of times by inserting multiple £1 coins, or none at all. Both of course (!) and why not (?) mean that

whatever follows it can be used any number of times, or even never used at all.

3.2 Statics

Propositions	A	$::=$	X	variable
			$X^\perp$	variable dual
			$A \otimes B$	tensor
			$A \parr B$	par
			$A \oplus B$	plus
			$A \,\&\, B$	with
			$!A$	of course
			$?A$	why not
			1	tensor unit
			$\perp$	par unit
			0	plus unit
			$\top$	with unit

Figure 3.1: Propositions for Classical Linear Logic

Now we have an intuition for how propositions work for CLL, we can adapt them to fit the new context with session types. Session types help us dictate exactly how we want processes to communicate with each other. They can describe patterns of communication, for example; first one process will send and the other will receive, then the second process will send and the first will receive. This allows us to create communicating processes which do not suffer from deadlock or endless waiting.

Tensor, $A \otimes B$ represents multiplicative conjunction and means output A, then continue as B. The dual to tensor is par which represents multiplicative disjunction. We would read $A \parr B$ as input A, then continue as B. We have plus for additive disjunction so $A \oplus B$ means select either A or B. Dual to this we have with, $A \,\&\, B$ meaning offer a choice between A and B. Our exponential component consists of of course, $!A$ which means we have a server which can accept many copies of A, and why not, $?A$ means we have a client who may request many copies of A.

Classical linear logic naturally lends itself to parallelism, so we present the rules of CLL alongside a process calculus similar to π calculus. We use round brackets, to show input, and square brackets to show output. A process P can take on a few forms. Link says that we can connect two channels in such a way that any message delivered to y will be forwarded over x and vice verca. For an output such as $y[x].\,(P \mid Q)$ we would output x on channel y then continue as P and Q in parallel. We have that y is bound in P, but is not bound in Q. For the input $y(x).\,P$, we input x on channel y then continue as P where y is bound

$$(A \otimes B)^\perp = A^\perp \parr B^\perp \qquad (A \parr B)^\perp = A^\perp \otimes B^\perp$$
$$(A \oplus B)^\perp = A^\perp \,\&\, B^\perp \qquad (A \,\&\, B)^\perp = A^\perp \oplus B^\perp$$
$$(!A)^\perp = ?A^\perp \qquad (?A)^\perp = !A^\perp$$
$$1^\perp = \perp \qquad \perp^\perp = 1$$
$$0^\perp = \top \qquad \top^\perp = 0$$

Figure 3.2: Duals for Classical Linear Logic Propositions

$P ::=$	
$P \mid Q$	parallel composition
$x \leftrightarrow y$	link x with y
$y[x].\, P \mid Q$	output x on channel y
$y(x).\, P$	input x on channel y
$x[\mathsf{inl}].\, P$	left selection
$x[\mathsf{inr}].\, P$	right selection
$x.\mathsf{case}(P; Q)$	choice
$\nu x.\, \{P \mid Q\}$	connect on channel x
$?y[x].\, P$	client request
$!y(x).\, P$	server accept
$y[\,].\, P$	empty output
$y(\,).\, P$	empty input
$x.\mathsf{case}()$	empty choice
$P[x/y]$	substitution

Figure 3.3: Process Calculus for Classical Linear Logic

in P. In $\nu x.\, \{P \mid Q\}$, we connect processes P and Q on channel x where x is bound in both P and Q. Server accept is just like input so for $!y(x).\, P$, we would input x on channel $!y$ then continue as P where x is bound in P. Similarly, for $?y[x].\, P$, we ouput x on channel $?y$ then continue as P where x is bound in P.

We can now combine our process calculus with our propositions for CLL and present the rules for classical linear logic alongside their corresponding processes. Our typing judgements have the form $P \vdash \Gamma, x : A$ where P is a process, Γ is a type environment, x is a channel, and A is a CLL proposition. This typing judgement means we have some process P communicating along channel x obeying protocol A. Processes can communicate over multiple channels following different protocols, and as we saw in 3.3 the processes may also take on varying forms.

The Axiom states that if we have some variable A, then we also have its dual $A^{\perp}$. For the process calculus, we see that if we have two channels following dual protocols then any input along x is sent as output along y, and vice versa. The Cut rule allows us to connect two processes following dual protocols together. As the protocols are dual, any transmissions and selections over one correspond with receives and choices over the other. The communication over Γ and Δ are distinct so the processes P and Q can only communicate over the shared channel x which ensures that processes cannot get stuck.

The Tensor rule outputs a fresh channel x along y, then continues as P and Q in parallel. As P and Q communicate over different channels, we have disjoint concurrency so the processes cannot communicate with each other. The new process, $P \mid Q$ communicates over channel y. The rule Par inputs A, then continues as B. This is known as connected concurrency as P can communicate along both x and y.

The rule Plus-L indicates left selection, and similarly Plus-R indicates right selection. The process $x[\mathsf{inl}].\, P$ obeys protocol $A \oplus B$ by requesting the left option from a choice sent along x. The process for $x[\mathsf{inr}].\, P$ is symmetric. The With rule offers a choice between processes P and Q. The new process $x.\mathsf{case}(P; Q)$ will receive a selection over channel x and execute either P or Q accordingly.

Weakening lets us consider a process which doesn't communicate or follow a protocol to be a process which communicates along a channel x with protocol ?A. This is the rule for having no clients. If a process P communicates along two channels, both following the same protocol ?A, then we can use Contraction to substitute one channel for another so P communicates along only one channel following protocol ?A. This rules lets us pool multiple clients so they communicate over just one channel. Dereliction corresponds to a single client and allows us to output a fresh channel x along y then execute the process

$$\textsc{Axiom}\ \frac{}{x \leftrightarrow y \vdash x : A^{\perp}, y : A} \qquad \textsc{Cut}\ \frac{P \vdash \Gamma, x : A^{\perp} \qquad Q \vdash \Delta, x : A}{\nu x.\, \{P \mid Q\} \vdash \Gamma, \Delta}$$

$$\textsc{Tensor}\ \frac{P \vdash \Gamma, x : A \qquad Q \vdash \Delta, y : B}{y[x].\, (P \mid Q) \vdash \Gamma, \Delta, y : A \otimes B} \qquad \textsc{Par}\ \frac{P \vdash \Gamma, x : A, y : B}{y(x).\, P \vdash \Gamma, y : A \parr B}$$

$$\textsc{Plus-L}\ \frac{P \vdash \Gamma, x : A}{x[\mathsf{inl}].\, P \vdash \Gamma, x : A \oplus B} \qquad \textsc{Plus-R}\ \frac{P \vdash \Gamma, x : B}{x[\mathsf{inr}].\, P \vdash \Gamma, x : A \oplus B} \qquad \textsc{With}\ \frac{P \vdash \Gamma, x : A \qquad Q \vdash \Gamma, x : B}{x.\mathsf{case}(P; Q) \vdash \Gamma, x : A \,\&\, B}$$

$$\textsc{Weakening}\ \frac{P \vdash \Gamma}{P \vdash \Gamma, x : ?A} \qquad \textsc{Contraction}\ \frac{P \vdash \Gamma, x : ?A, y : ?A}{P[x/y] \vdash \Gamma, x : ?A} \qquad \textsc{Dereliction}\ \frac{P \vdash \Gamma, x : A}{?y[x].\, P \vdash \Gamma, y : ?A}$$

$$\textsc{Promotion}\ \frac{P \vdash ?\Gamma, x : A}{!y(x).\, P \vdash ?\Gamma, y : !A}$$

$$\frac{}{x[\,].\, P \vdash x : 1} \qquad \frac{P \vdash \Gamma}{x(\,).\, P \vdash \Gamma, x : \bot} \qquad \frac{}{x.\mathsf{case}() \vdash \Gamma, x : \top}$$

Figure 3.4: Static Rules for Classical Processes

P. Promotion receives x along the channel y and creates a fresh copy of P to execute. Any other channels used by P must follow protocols of the form ?B to ensure typing is respected when we replicate P.

3.3 Dynamics

Just like in the simply typed lambda calculus, there are dynamic rules for classical processes. These dynamic rules are split into two distinct sections: cut reduction, and cut elimination. We have already seen the Cut rule and the dynamics for CP make use of this rule to simplify processes. Before venturing into cut reduction, we have some structural cut equivalences. Swap illustrates that cut is a symmetric relation, and Assoc allows us to reorder processes within cuts.

$$\textsc{Swap}\ \frac{P \vdash \Gamma, x : A \qquad Q \vdash \Delta, x : A^{\perp}}{\nu x.\, \{P \mid Q\} \vdash \Gamma, \Delta} \equiv \frac{Q \vdash \Delta, x : A^{\perp} \qquad P \vdash \Gamma, x : A}{\nu x.\, \{P \mid Q\} \vdash \Gamma, \Delta}$$

$$\textsc{Assoc}\ \frac{\dfrac{P \vdash \Gamma, x : A \qquad Q \vdash \Delta, X : A^{\perp}, y : B}{\nu x.\, \{P \mid Q\} \vdash \Gamma, \Delta, y : B} \qquad R \vdash \Theta, y : B^{\perp}}{\nu y.\, \{\nu x.\, \{P \mid Q\} \mid R\} \vdash \Gamma, \Delta, \Theta} \equiv$$

$$\frac{P \vdash \Gamma, x : A \qquad \dfrac{Q \vdash \Delta, x : A^{\perp}, y : B \qquad R \vdash \Theta, y : B^{\perp}}{\nu y.\, \{Q \mid R\} \vdash \Delta, \Theta, x : A^{\perp}}}{\nu x.\, \{P \mid \nu y.\, \{Q \mid R\}\} \vdash \Gamma, \Delta, \Theta}$$

Figure 3.5: Structural Cut Equivalences for Classical Processes

As in the λ-calculus, we have the notions of progress and preservation for CP. We use the definitions of pregress and preservation presented by Kokke, Montesi, and Peressotti [7]. We will state these here, along with the definitions of actions and canonical forms which are needed to understand progress.

Actions: A process P acts on x whenever x is free in the outermost term constructor of P. A process P is an action if it acts on some channel x. For example, $x(y).\,P$ acts on x but not on y.

Canonical Forms: A process P is in canonical forms if $P \equiv \nu x_1.\,\{P_1 \mid \dots \nu x_n.\,\{P_n \mid P_{n+1}\} \dots\}$ such that no process P_i is a cut; no process P_i a link acting on a bound channel x_i; and no two processes P_i and P_j are acting on the same bound channel x_i.

Preservation: If $P \vdash \Gamma$ and $P \Longrightarrow Q$, then $Q \vdash \Gamma$.

Progress: If $P \vdash \Gamma$ then either P is in canonical forms, or there exists a process Q such that $P \Longrightarrow Q$.

3.3.1 Cut Reduction

Cut reduction shows us how to handle specific pairs of processes within a cut. So for some pair of processes which are connected via a cut, the reduction rules illustrate which parts of each process is used for communication. It also serves as a proof of preservation for CP.

$$\textsc{AxCut}\quad \frac{\dfrac{}{x \leftrightarrow y \vdash x : A^{\perp}, y : A} \qquad P \vdash \Gamma, x : A}{\nu x.\,\{x \leftrightarrow y \mid P\} \vdash \Gamma, y : A} \Longrightarrow_{axCut} \quad P[y/x] \vdash \Gamma, y : A$$

$$\frac{\dfrac{P \vdash \Gamma, x : A^{\perp} \qquad Q \vdash \Delta, y : B^{\perp}}{y[x].\,(P \mid Q) \vdash \Gamma, \Delta, y : A^{\perp} \otimes B^{\perp}} \qquad \dfrac{R \vdash \Theta, x : A, y : B}{y(x).\,R \vdash \Theta, y : A \parr B}}{\nu y.\,\{y[x].\,(P \mid Q) \mid y(x).\,R\} \vdash \Gamma, \Delta, \Theta} \Longrightarrow_{\beta_{\otimes\parr}}$$

$$\frac{\dfrac{P \vdash \Gamma, x : A^{\perp} \qquad R \vdash \Theta, x : A, y : B}{\nu x.\,\{P \mid Q\} \vdash \Gamma, \Theta, y : B} \qquad Q \vdash \Delta, y : B^{\perp}}{\nu y.\,\{\nu x.\,\{P \mid R\} \mid Q\} \vdash \Gamma, \Delta, \Theta}$$

$$\frac{\dfrac{P \vdash \Gamma, x : A}{x[\mathsf{inl}].\,P \vdash \Gamma, x : A \oplus B} \qquad \dfrac{Q \vdash \Delta, x : A^{\perp} \qquad R \vdash \Theta, x : B^{\perp}}{x.\mathsf{case}(Q; R) \vdash \Delta, x : A \,\&\, B}}{\nu x.\,\{x[\mathsf{inl}].\,P \mid x.\mathsf{case}(Q; R)\} \vdash \Gamma, \Delta} \Longrightarrow_{\beta_{\oplus\&}}$$

$$\frac{P \vdash \Gamma, x : A \qquad Q \vdash \Delta, x : A^{\perp}}{\nu x.\,\{P \mid Q\} \vdash \Gamma, \Delta}$$

Figure 3.6: Cut Reduction Rules for Classical Processes I

AxCut is a structural cut reduction and shows that we can perform a substitution when we have a link cut with a process sharing one channel with the link. The general rule is that if we have an input such as $x(y).\,P$ or $x.\mathsf{case}(Q; R)$ on one side of a cut, and an output such as $x[y].\,Q$ or $x[\mathsf{inr}].\,P$ on the other, then cutting them together allows us to connect the subprocesses directly together. So, $\nu x.\,\{x(y).\,P \mid x[y].\,Q\} \Longrightarrow \nu x.\,\{P \mid Q\}$ and $\nu x.\,\{x[\mathsf{inr}].\,P \mid x.\mathsf{case}(Q; R)\} \Longrightarrow \nu x.\,\{P \mid R\}$.

$$\frac{\dfrac{P \vdash ?\Gamma, x : A}{!y(x).\, P \vdash ?\Gamma, y : !A} \qquad \dfrac{Q \vdash \Delta, x : A^{\perp}}{?y[x].\, Q \vdash \Delta, y : ?A^{\perp}}}{\nu y.\, \{!y(x).\, P \mid ?y[x].\, Q\} \vdash ?\Gamma, \Delta} \Longrightarrow_{\beta_{!?}} \frac{P \vdash ?\Gamma, x : A \qquad Q \vdash \Delta, x : A^{\perp}}{\nu x.\, \{P \mid Q\} \vdash ?\Gamma, \Delta}$$

$$\frac{\dfrac{P \vdash ?\Gamma, x : A}{!y(x).\, P \vdash ?\Gamma, y : !A} \qquad \dfrac{Q \vdash \Delta}{Q \vdash \Delta, x : ?A^{\perp}}}{\nu y.\, \{!y(x).\, P \mid Q\} \vdash ?\Gamma, \Delta} \Longrightarrow_{\beta_{!W}} \frac{Q \vdash \Delta}{Q \vdash ?\Gamma, \Delta}$$

$$\frac{\dfrac{P \vdash ?\Gamma, x : A}{!y(x).\, P \vdash ?\Gamma, y : !A} \qquad \dfrac{Q \vdash \Delta, y : ?A, z : ?A}{Q[y/z] \vdash \Delta, y : ?A}}{\nu y.\, \{!y(x).\, P \mid Q[y/z]\} \vdash ?\Gamma, \Delta} \Longrightarrow_{\beta_{!C}}$$

$$\frac{\dfrac{P \vdash ?\Gamma, x : A}{!y(x).\, P \vdash ?\Gamma, y : !A} \qquad \dfrac{\dfrac{P' \vdash ?\Gamma', w : A}{!z(w).\, P \vdash ?\Gamma', z : !A} \qquad Q \vdash \Delta, y : ?A^{\perp}, z : ?A^{\perp}}{\nu z.\, \{!z(w).\, P \mid Q\} \vdash ?\Gamma', \Delta, x : ?A^{\perp}}}{\dfrac{\nu y.\, \{!y(x).\, P \mid \nu z.\, \{!z(w).\, P' \mid Q\}\} \vdash ?\Gamma, ?\Gamma', \Delta}{\nu y.\, \{!y(x).\, P \mid \nu z.\, \{!z(y).\, P \mid Q\}\} \vdash ?\Gamma, \Delta}}$$

$$\frac{\dfrac{P \vdash \Gamma}{x(\,).\, P \vdash \Gamma, x : \perp} \qquad \dfrac{}{x[\,].\, Q \vdash x : 1}}{\nu x.\, \{x(\,).\, P \mid x[\,].\, Q\} \vdash \Gamma} \Longrightarrow_{\beta_{1\perp}} P \vdash \Gamma$$

Figure 3.7: Cut Reduction Rules for Classical Processes II

3.3.2 Cut Elimination

Cut elimination, also called commuting conversions, illustrate how we can push cuts into communication protocols. So, if we have a cut $\nu x.\, \{P \mid Q\}$ and the process $P = y(z).\, R$ is communicating on a different channel, y, then we can push the cut into the communicating process to get a new process $y(z).\, \nu x.\, \{R \mid Q\}$.

$$\frac{\dfrac{P \vdash \Gamma, y : A, z : C \qquad Q \vdash \Delta, x : B}{x[y].\,(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B, z : C} \qquad R \vdash \Theta, z : C^{\perp}}{\nu z.\,\{x[y].\,(P \mid Q) \mid R\} \vdash \Gamma, \Delta, \Theta, x : A \otimes B} \Longrightarrow_{\kappa_{\otimes 1}}$$

$$\frac{Q \vdash \Delta, x : B \qquad \dfrac{P \vdash \Gamma, y : A, z : C \qquad R \vdash \Theta, z : C^{\perp}}{\nu z.\,\{P \mid R\} \vdash \Gamma, \Theta, y : A}}{x[y].\,(\nu z.\,\{P \mid R\}) \mid Q \vdash \Gamma, \Delta, \Theta, x : A \otimes B}$$

$$\frac{\dfrac{P \vdash \Gamma, x : A, \qquad Q \vdash \Delta, x : B, z : C}{x[y].\,(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B, z : C} \qquad R \vdash \Theta, z : C^{\perp}}{\nu z.\,\{x[y].\,(P \mid Q) \mid R\} \vdash \Gamma, \Delta, \Theta, x : A \otimes B} \Longrightarrow_{\kappa_{\otimes 2}}$$

$$\frac{P \vdash \Gamma, y : A \qquad \dfrac{Q \vdash \Delta, x : B, z : C \qquad R \vdash \Theta, z : C^{\perp}}{\nu z.\,\{Q \mid R\} \vdash \Delta, \Theta, x : B}}{x[y].\,(P \mid \nu z.\,\{Q \mid R\}) \vdash \Gamma, \Delta, \Theta, x : A \otimes B}$$

$$\frac{\dfrac{P \vdash \Gamma, y : A, x : B, z : C}{x(y).\,P \vdash \Gamma, x : A \parr B, z : C} \qquad Q \vdash \Delta, z : C^{\perp}}{\nu z.\,\{x(y).\,P \mid Q\} \vdash \Gamma, \Delta, x : A \parr B} \Longrightarrow_{\kappa_{\parr}} \frac{\dfrac{P \vdash \Gamma, y : A, x : B, z : C \qquad Q \vdash \Delta, z : C^{\perp}}{\nu z.\,\{P \mid Q\} \vdash \Gamma, \Delta, y : A, x : B}}{x(y).\,\nu z.\,\{P \mid Q\} \vdash \Gamma, \Delta, x : A \parr B}$$

$$\frac{\dfrac{P \vdash \Gamma, x : A, z : C \qquad Q \vdash \Gamma, x : B, z : C}{x.\mathsf{case}(P; Q) \vdash \Gamma, x : A \,\&\, B, z : C} \qquad R \vdash \Delta, z : C^{\perp}}{\nu z.\,\{x.\mathsf{case}(P; Q) \mid R\} \vdash \Gamma, \Delta, x : A \,\&\, B} \Longrightarrow_{\kappa_{\&}}$$

$$\frac{\dfrac{P \vdash \Gamma, x : A, z : C \qquad R \vdash \Delta, z : C^{\perp}}{\nu z.\,\{P \mid R\} \vdash \Gamma, \Delta, x : A} \qquad \dfrac{Q \vdash \Gamma, x : B, z : C \qquad R \vdash \Delta, z : C^{\perp}}{\nu z.\,\{Q \mid Q\} \vdash \Gamma, \Delta, x : B}}{x.\mathsf{case}(\nu z.\,\{P \mid R\}; \nu z.\,\{Q \mid R\}) \vdash \Gamma, \Delta, x : A \,\&\, B}$$

$$\frac{\dfrac{P \vdash \Gamma, x : A, z : C}{x[\mathsf{inl}].\,P \vdash \Gamma, x : A \oplus B, z : C} \qquad Q \vdash \Delta, z : C^{\perp}}{\nu z.\,\{x[\mathsf{inl}].\,P \mid Q\} \vdash \Gamma, \Delta, x : A \oplus B} \Longrightarrow_{\kappa_{\oplus}} \frac{\dfrac{P \vdash \Gamma, x : A, z : C \qquad Q \vdash \Delta, z : C^{\perp}}{\nu z.\,\{P \mid Q\} \vdash \Gamma, \Delta, x : A}}{x[\mathsf{inl}].\,\nu z.\,\{P \mid Q\} \vdash \Gamma, \Delta, x : A \oplus B}$$

Figure 3.8: Cut Elimination for Classical Processes I

$$\dfrac{\dfrac{P \vdash ?\Gamma, y : A, z : ?C}{!x(y).\, P \vdash ?\Gamma, x : !A, z : ?C} \qquad Q \vdash ?\Delta, z : !C^{\perp}}{\nu z.\, \{!x(y).\, P \mid Q\} \vdash ?\Gamma, ?\Delta, x : !A} \Longrightarrow_{\kappa_!} \dfrac{\dfrac{P \vdash ?\Gamma, y : A, z : ?C \qquad Q \vdash ?\Delta, z : !C^{\perp}}{\nu z.\, \{P \mid Q\} \vdash ?\Gamma, ?\Delta, y : A}}{!x(y).\, \nu z.\, \{P \mid Q\} \vdash ?\Gamma, ?\Delta, x : !A}$$

$$\dfrac{\dfrac{P \vdash \Gamma, y : A, z : C}{?x[y].\, P \vdash \Gamma, x : ?A, z : C} \qquad Q \vdash \Delta, z : C^{\perp}}{\nu z.\, \{?x[y].\, P \mid Q\} \vdash \Gamma, \Delta, x : ?A} \Longrightarrow_{\kappa_?} \dfrac{\dfrac{P \vdash \Gamma, y : A, z : C \qquad Q \vdash \Delta, z : C^{\perp}}{\nu z.\, \{P \mid Q\} \vdash \Gamma, \Delta, y : A}}{?x[y].\, \nu z.\, \{P \mid Q\} \vdash \Gamma, \Delta, x : ?A}$$

$$\dfrac{\dfrac{P \vdash \Gamma, z : C}{x(\,).\, P \vdash \Gamma, x : \perp, z : C} \qquad Q \vdash \Delta, z : C^{\perp}}{\nu z.\, \{x(\,).\, P \mid Q\} \vdash \Gamma, \Delta, x : \perp} \Longrightarrow_{\kappa_\perp} \dfrac{\dfrac{P \vdash \Gamma, z : C \qquad Q \vdash \Delta, z : C^{\perp}}{\nu z.\, \{P \mid Q\} \vdash \Gamma, \Delta}}{x(\,).\, \nu z.\, \{P \mid Q\} \vdash \Gamma, \Delta, x : \perp}$$

$$\dfrac{\dfrac{}{x.\mathsf{case}() \vdash \Gamma, x : \top, z : C} \qquad Q \vdash \Delta, z : C^{\perp}}{\nu z.\, \{x.\mathsf{case}(;) \mid Q\} \vdash \Gamma, \Delta, x : \top} \Longrightarrow_{\kappa_\top} \dfrac{}{x.\mathsf{case}() \vdash \Gamma, \Delta, x : \top}$$

Figure 3.9: Cut Elimination for Classical Processes II

Chapter 4

Translation

Here we present a translation from the simply typed lambda calculus into classical linear logic. First we will give the direct translation of each type, then a detailed derivation of how we got there along with an explanation of what exactly is happening in our translated protocol. We make use of the $^\circ$ symbol to denote translation of a type. In some of the translations, we will also use the † symbol which is simply a renaming of the translation and acts in the same way as $^\circ$. On the left hand side we have the lambda calculus type, and on the right we have its classical linear logic proposition.

$$\begin{aligned}(1)^\circ &= \top \\ (A+B)^\circ &= !A^\circ \oplus !B^\circ \\ (A \times B)^\circ &= A^\circ \,\&\, B^\circ \\ (A \to B)^\circ &= (!A^\circ)^\perp \parr B^\circ\end{aligned}$$

Figure 4.1: Translation of Types

The translations of 1 and $(A \times B)$ are as one would expect. The exponentials in the translation of $(A+B)$ indicate that we may select multiple times from an option we are sent. The translation of $(A \to B)$ makes use negation since we use some A to acquire some B. The exponential $!A^\circ$ indicates that we can use our function multiple times, rather than just once. If we had just $A^{\circ\perp} \parr B^\circ$ on the CLL side, we would only be able to use the function once as we only have access to one $A^{\circ\perp}$.

The translation of some environment Γ is as follows:

$$\begin{aligned}\Gamma^\circ& \\ (\cdot)^\circ &= \cdot \\ (\Gamma, x : A)^\circ &= \Gamma^\circ, x : (!A^\circ)^\perp \\ &= \Gamma^\circ, x : ?(A^\circ)^\perp\end{aligned}$$

Figure 4.2: Translation of Contexts

If we have a term $x : A$ from the lambda calculus, translates to $x : (!A^\circ)^\perp$. The negation occurs because contexts in the lambda calculus occur on the left hand side, but we only have right-handed sequents in classical linear logic. More interestingly, we have the exponentiation, $!A^\circ$ rather than just A°. This is because in the lambda calculus we may use a protocol as many times as we wish if it occurs in the context, but in classical linear logic we may only use something exactly as many times as it occurs. Hence, we have the exponentiation $(!A^\circ)$ to indicate that we may use A° as many times as we desire. There is also an implicit ?before any translated context since we can choose not to use a context when working in the λ-calculus.

We will need a derivable rule, Depar which takes a concurrent process $A \parr B$ communicating along a channel z and disconnects them to give us both A and B on separate channels. We achieve this by creating copies of the individual components $x : A$ and $y : B$ along with their duals $w : A^\perp, z : B^\perp$, each of which communicate along separate channels and can forward information to their dual. We then output w along z allowing the protocols to continue in parallel. This gives us the dual, $z : A^\perp \otimes B^\perp$, to our original process $z : A \parr B$ and we can connect on channel z to give us the disconnected processes A and B.

$$\dfrac{\dfrac{w \leftrightarrow x \vdash x : A, w : A^\perp \qquad y \leftrightarrow z \vdash y : B, z : B^\perp}{z[w].\,(w \leftrightarrow x \mid y \leftrightarrow z) \vdash x : A, y : B, z : A^\perp \otimes B^\perp} \qquad \overline{P \vdash \Gamma, z : A \parr B}}{\nu z.\,\{z[w].\,(w \leftrightarrow x \mid y \leftrightarrow z) \mid P\} \vdash \Gamma, x : A, y : B} \quad \rightsquigarrow \quad \dfrac{P \vdash \Gamma, z : A \parr B}{xy.\mathsf{depar}\{P\} \vdash \Gamma, x : A, y : B}\ \textsc{Depar}$$

It is interesting to note that we have an inversion of polarity from the lambda calculus to classical linear logic. All type constructors in the lambda calculus become destructors in CLL and vice verca.

For our translation T-Var with respect to y, we begin with the CLL axiom to give us two channels following dual protocols A° and $(A^\circ)^\perp$ which share information with each other via forwarding. We then use dereliction to request x, and therefore the protocol $?(A^\circ)^\perp$ on a new channel z. Finally we use weakening to acquire an environment Γ° which we do not use. This gives us a typing judgement which is equivalent to the variable judgement in the lambda calculus. So, our final judgement says link $x : (A^\circ)^\perp$ with $y : A^\circ$, then request $x : (A^\circ)^\perp$ on a new channel z.

For T-Unit we simply use the empty case statement along with the unit for with. So, we create a channel x with no protocols communicating along it.

The translation for T-In-l is fairly straightforward. We begin with a process communicating along channel y following protocol A° and use promotion to input y along channel x to acquire the protocol $!A^\circ$. We then use plus-l indicating we have selected the left choice of two protocols communicating on channel x then execute the process $[\![M]\!]_x$. The translation of T-In-r is equivalent, but we select the right choice.

For the translation T-Case, we use the anti-Barendregt convention and conveniently name variables in such a way that we do not have to rename them in the future. This is why we have $x : A$ and $x : B$ rather than the expected $y : B$. As in the lambda calculus, we start with three separate processes, each with the same environment. Our first step is to use the with rule to offer a choice between protocols $(!A^\circ)^\perp$ and $(!B^\circ)^\perp$ on channel x. Then we use the cut rule to connect the protocols on channel x. We then use contraction to remove copies of $y : C^\circ$ and allow our process to communicate along one channel.

For T-Prod we begin with two processes which communicate along the same channel x following different protocols. Then we use the with rule to offer a choice between protocols A° and B° along the same channel x.

The translation T-Prj-l begins with the two dual protocols then requests the left option from a choice sent along y. We then use the cut rule to connect the dual protocols $A^{\circ\perp} \oplus B^{\circ\perp}$ and $A^\circ \,\&\, B^\circ$ on channel y and acquire the left projection A° of the pair $A^\circ \,\&\, B^\circ$. The translation of T-Prj-r is analogous.

Our translation T-Lam with respect to y begins with a process following two protocols which commmunicate along serparate channels. Since we have $\Gamma, x : A \vdash$ on the λ- calculus side, we translate this to $(!A^\circ)^\perp$. We then use the par rule to input $(!A^\circ)^\perp$ along channel y then continue as B°. The'of course' indicates that we can apply our function $[\![M]\!]_y$ to multiple inputs of the form $(A^\circ)^\perp$.

Finally, T-App begins with a process following the protocol $!A^{\circ\perp} \parr B^\circ$. We use our derivable rule Depar to disconnect the protocols components and allow them to communicate along separate channels. On the other side we have a process following protocol A° which communicate along channel u with the environment $\Gamma^\dagger$ which is just a copy of the environment Γ°. We use promotion to input u along channel

$$\left[\!\!\left[\dfrac{}{\Gamma, x : A \vdash x : A}\ \text{T-Var}\right]\!\!\right]_y \stackrel{\text{def}}{=} \dfrac{\dfrac{}{z \leftrightarrow y \vdash z : (A^\circ)^\perp, y : A^\circ}}{?x[z].\, z \leftrightarrow y \vdash x : ?(A^\circ)^\perp, y : A^\circ}$$

$$\left[\!\!\left[\dfrac{}{\Gamma \vdash \langle\rangle : 1}\ \text{T-Unit}\right]\!\!\right]_x \stackrel{\text{def}}{=} \dfrac{}{x.\mathsf{case}() \vdash \Gamma^\circ, x : \top}$$

$$\left[\!\!\left[\dfrac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{inl}(M) : A + B}\ \text{T-In-L}\right]\!\!\right]_x \stackrel{\text{def}}{=} \dfrac{\dfrac{[\![M]\!]_y \vdash \Gamma^\circ, y : A^\circ}{!x(y).\, [\![M]\!]_y \vdash \Gamma^\circ, x : !A^\circ}}{x[\mathsf{inl}].\, !x(y).\, [\![M]\!]_y \vdash \Gamma^\circ, x : !A^\circ \oplus !B^\circ}$$

$$\left[\!\!\left[\dfrac{\Gamma \vdash M : B}{\Gamma \vdash \mathsf{inr}(M) : A + B}\ \text{T-In-R}\right]\!\!\right]_x \stackrel{\text{def}}{=} \dfrac{\dfrac{[\![M]\!]_y \vdash \Gamma^\circ, y : B^\circ}{!x(y).\, [\![M]\!]_y \vdash \Gamma^\circ, x : !B^\circ}}{x[\mathsf{inr}].\, !x(y).\, [\![M]\!]_y \vdash \Gamma^\circ, x : !A^\circ \oplus !B^\circ}$$

$$\left[\!\!\left[\dfrac{\Gamma \vdash M : A + B \qquad \Gamma, x : A \vdash P : C \qquad \Gamma, x : B \vdash Q : C}{\Gamma \vdash \mathsf{case}(M; x.\, P; x.\, Q) : C}\ \text{T-Case}\right]\!\!\right]_y \stackrel{\text{def}}{=}$$

$$\dfrac{\dfrac{\dfrac{[\![M]\!]_x \vdash \Gamma^\circ, x : !A^\circ \oplus !B^\circ \qquad [\![P]\!]_y \vdash \Gamma^\circ, y : C^\circ, x : (!A^\circ)^\perp \qquad [\![Q]\!]_y \vdash \Gamma^\circ, y : C^\circ, x : (!B^\circ)^\perp}{[\![M]\!]_x \vdash \Gamma^\circ, x : !A^\circ \oplus !B^\circ \qquad x.\mathsf{case}([\![P]\!]_y; [\![Q]\!]_y) \vdash \Gamma^\circ, y : C^\circ, x : (!A^\circ)^\perp \,\&\, (!B^\circ)^\perp}}{\nu x.\, \{[\![M]\!]_x \mid x.\mathsf{case}([\![P]\!]_y; [\![Q]\!]_y)\} \vdash \Gamma^\circ, y : C^\circ}}{\nu x.\, \{[\![M]\!]_x \mid x.\mathsf{case}([\![P]\!]_y; [\![Q]\!]_y)\} \vdash \Gamma^\circ, y : C^\circ}$$

$$\left[\!\!\left[\dfrac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N\rangle : A \times B}\ \text{T-Prod}\right]\!\!\right]_x \stackrel{\text{def}}{=} \dfrac{[\![M]\!]_x \vdash \Gamma^\circ, x : A^\circ \qquad [\![N]\!]_x \vdash \Gamma^\circ, x : B^\circ}{x.\mathsf{case}([\![M]\!]_x; [\![N]\!]_x) \vdash \Gamma^\circ, x : A^\circ \,\&\, B^\circ}$$

$$\left[\!\!\left[\dfrac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_1(M) : A}\ \text{T-Prj-L}\right]\!\!\right]_x \stackrel{\text{def}}{=} \dfrac{\dfrac{}{x \leftrightarrow y \vdash x : A^\circ, y : (A^\circ)^\perp}}{y[\mathsf{inl}].\, x \leftrightarrow y \vdash x : A^\circ, y : (A^\circ)^\perp \oplus (B^\circ)^\perp} \qquad \dfrac{}{[\![M]\!]_y \vdash \Gamma^\circ, y : A^\circ \,\&\, B^\circ}$$
$$\nu y.\, \{y[\mathsf{inl}].\, y \leftrightarrow x \mid [\![M]\!]_y\} \vdash \Gamma^\circ, x : A^\circ$$

$$\left[\!\!\left[\dfrac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_2(M) : B}\ \text{T-Prj-R}\right]\!\!\right]_x \stackrel{\text{def}}{=} \dfrac{\dfrac{}{x \leftrightarrow y \vdash x : B^\circ, y : (B^\circ)^\perp}}{y[\mathsf{inr}].\, x \leftrightarrow y \vdash x : B^\circ, y : (A^\circ)^\perp \oplus (B^\circ)^\perp} \qquad \dfrac{}{[\![M]\!]_y \vdash \Gamma^\circ, y : A^\circ \,\&\, B^\circ}$$
$$\nu y.\, \{y[\mathsf{inr}].\, y \leftrightarrow x \mid [\![M]\!]_y\} \vdash \Gamma^\circ, x : B^\circ$$

$$\left[\!\!\left[\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.\, M : A \to B}\ \text{T-Lam}\right]\!\!\right]_y \stackrel{\text{def}}{=} \dfrac{[\![M]\!]_y \vdash \Gamma^\circ, x : (!A^\circ)^\perp, y : B^\circ}{y(x).\, [\![M]\!]_y \vdash \Gamma^\circ, y : (!A^\circ)^\perp \parr B^\circ}$$

$$\left[\!\!\left[\dfrac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ \text{T-App}\right]\!\!\right]_y \stackrel{\text{def}}{=}$$

$$\dfrac{\dfrac{\dfrac{[\![M]\!]_z \vdash \Gamma^\circ, z : (!A^\circ)^\perp \parr B^\circ}{xy.\mathsf{depar}\{[\![M]\!]_z\} \vdash \Gamma^\circ, x : (!A^\circ)^\perp, y : B^\circ} \qquad \dfrac{[\![N]\!]_u \vdash \Gamma^\dagger, u : A^\circ}{!x^\dagger(u).\, [\![N]\!]_u \vdash \Gamma^\dagger, x^\dagger : !A^\circ}}{\nu x.\, \{xy.\mathsf{depar}\{[\![M]\!]_z\} \mid !x^\dagger(u).\, [\![N]\!]_u\} \vdash \Gamma^\circ, \Gamma^\dagger, y : B^\circ}}{\nu x.\, \{xy.\mathsf{depar}\{[\![M]\!]_z\} \mid !x(u).\, [\![N]\!]_u\} \vdash \Gamma^\circ, y : B^\circ}$$

Figure 4.3: Translation from the λ-Calculus into Classical Processes

$x^\dagger$ (a copy of the channel x) to acquire the protocol $!A^\circ$. We then connect the dual protocols $!A^{\circ\perp}$ and $!A^\circ$ on channel x which gives us a process which communicate along channel y following protocol B°. Finally, we use contraction to merge Γ° and its copy $\Gamma^\dagger$ into a single environment.

4.1 Proof of Simulation

Now we have a translation from the λ-calculus into CLL, we want to ensure it is correct. We will do this using a proof of simulation to show that reduction in the λ-calculus maps to reduction in our translation. More specifically, if $M \longmapsto M'$ then we must have that $[\![M]\!]_x \longmapsto [\![M']\!]_x$.

We proceed by induction on the dynamics for the λ-calculus (2.4) and use cut reduction (3.6, 3.7) on each the translation until we cannot reduce terms any further. However, before we can begin our proof, just like in the dynamics for the λ- calculus, we need the substitution lemma.

4.1.1 Substitution

Substitution: If we have some substitution $M[N/x]$ in the λ-calculus, then it's translation $[\![M[N/x]]\!]_y$ is equivalent to the following in classical processes: $\nu x.\{!x(z).[\![N]\!]_z \mid [\![M]\!]_y\}$.

We will now prove that this translation of substitution is correct by doing case analysis on the definition of substitution for the λ-calculus. We will need our cut reduction rules (3.6, 3.7), and our cut equivalences, both structural and otherwise (3.5, 3.8, 3.9). We begin each case by stating the substitution we are translating and what it should reduce to, then give the reduction process we used to get to the intended result.

Case: $[\![x[N/x]]\!]_y \stackrel{\text{def}}{=} [\![N]\!]_y$

$$\begin{array}{lll} [\![x[N/x]]\!]_y & \stackrel{\text{def}}{=} & \nu x.\{!x(z).[\![N]\!]_z \mid [\![x]\!]_y\} \\ & = & \nu x.\{!x(z).[\![N]\!]_z \mid ?x[z].\, z \leftrightarrow y\} \\ & \Longrightarrow_{\beta_{!?}} & \nu z.\{[\![N]\!]_z \mid z \leftrightarrow y\} \\ & \Longrightarrow_{axCut} & [\![N]\!]_z[y/z] \\ & = & [\![N]\!]_y \end{array}$$

For variable substitution, we simply extract the translation of x with respect to y, then use the $\beta_{!?}$ reduction rule as the cut occurs on the same channel as the input and output. We then use axCut to substitute y for z in the translation of N with respect to z and attain the desired result.

Case: $[\![w[N/x]]\!]_y \stackrel{\text{def}}{=} [\![w]\!]_y$

$$\begin{array}{lll} [\![w[N/x]]\!]_y & \stackrel{\text{def}}{=} & \nu x.\{!x(z).[\![N]\!]_z \mid [\![w]\!]_y\} \\ & = & \nu x.\{!x(z).[\![N]\!]_z \mid ?w[u].\, u \leftrightarrow y\} \\ & \Longrightarrow_{\beta_{!W}} & ?w[u].\, u \leftrightarrow y \\ & = & [\![w]\!]_y \end{array}$$

This follows a similar process to the previous case, except we use the $\beta_{!W}$ reduction rule since the cut occurs on a different channel to the output. This eliminates the cut and leaves us with just the translation of the variable w with respect to y.

Case: $[\![\langle\rangle[N/x]]\!]_y \stackrel{\text{def}}{=} [\![\langle\rangle]\!]_y$

$$\begin{array}{lll}
[\![\langle\rangle[N/x]]\!]_y & \stackrel{\text{def}}{=} & \nu x.\,\{!x(z).\,[\![N]\!]_z \mid [\![\langle\rangle]\!]_y\} \\
& = & \nu x.\,\{!x(z).\,[\![N]\!]_z \mid y.\mathsf{case}()\} \\
& \equiv_{swap} & \nu x.\,\{y.\mathsf{case}() \mid !x(z).\,[\![N]\!]_z\} \\
& \Longrightarrow_{\kappa_\top} & y.\mathsf{case}() \\
& = & [\![\langle\rangle]\!]_y
\end{array}$$

Here we simply use swap to have the processes within the cut switch positions then use a cut elimination rule to remove the latter half of the cut. This leaves us with the unit translated with respect to y as required.

Case: $[\![\mathsf{inl}(M)[N/x]]\!]_y \stackrel{\text{def}}{=} [\![\mathsf{inl}(M[N/x])]\!]_y$

$$\begin{array}{lll}
[\![\mathsf{inl}(M)[N/x]]\!]_y & \stackrel{\text{def}}{=} & \nu x.\,\{!x(z).\,[\![N]\!]_z \mid [\![\mathsf{inl}(M)]\!]_y\} \\
& = & \nu x.\,\{!x(z).\,[\![N]\!]_z \mid y[\mathsf{inl}].\,!y(w).\,[\![M]\!]_w\} \\
& \equiv_{swap} & \nu x.\,\{y[\mathsf{inl}].\,!y(w).\,[\![M]\!]_w \mid !x(z).\,[\![N]\!]_z\} \\
& \Longrightarrow_{\kappa_\oplus} & y[\mathsf{inl}].\,\nu x.\,\{!y(w).\,[\![M]\!]_w \mid !x(z).\,[\![N]\!]_z\} \\
& \Longrightarrow_{\kappa_!} & y[\mathsf{inl}].\,!y(w).\,\nu x.\,\{[\![M]\!]_w \mid !x(z).\,[\![N]\!]_z\} \\
& \equiv_{swap} & y[\mathsf{inl}].\,!y(w).\,\nu x.\,\{!x(z).\,[\![N]\!]_z \mid [\![M]\!]_w\} \\
& = & y[\mathsf{inl}].\,!y(w).\,[\![M[N/x]]\!]_w \\
& = & [\![\mathsf{inl}(M[N/x])]\!]_y
\end{array}$$

For this case, again we use swap to switch the positions of the processes within the cut, then we use our cut elimination rules $\kappa_\oplus$ and $\kappa_!$ to pull the subprocesses $y[\mathsf{inl}].$ and $!y(w).$ respectively out of the cut on x. Then we use swap again which gives us the required result. The case for $\mathsf{inr}(M)[N/x]$ is analogous.

Case: $[\![\mathsf{case}(M; w.\,P; w.\,Q)[N/x]]\!]_y \stackrel{\text{def}}{=} [\![\mathsf{case}(M[N/x]; w.\,P[N/x]; w.\,Q[N/x])]\!]_y$

$$\begin{array}{lll}
[\![\mathsf{case}(M; w.\,P; w.\,Q)[N/x]]\!]_y & \stackrel{\text{def}}{=} & \nu x.\,\{!x(z).\,[\![N]\!]_z \mid [\![\mathsf{case}(M; w.\,P; w.\,Q)]\!]_y\} \\
& = & \nu x.\,\{!x(z).\,[\![N]\!]_z \mid \nu w.\,\{[\![M]\!]_w \mid w.\mathsf{case}([\![P]\!]_y; [\![Q]\!]_y)\}\} \\
& \equiv_{assoc} & \nu w.\,\{\nu x.\,\{!x(z).\,[\![N]\!]_z \mid [\![M]\!]_w\} \mid w.\mathsf{case}([\![P]\!]_y; [\![Q]\!]_y)\} \\
& = & \nu w.\,\{\nu x.\,\{!x(z).\,[\![N]\!]_z \mid [\![M]\!]_w[u/x][x/u]\} \mid w.\mathsf{case}([\![P]\!]_y; [\![Q]\!]_y)\} \\
& \Longrightarrow_{\beta_{!C}} & \nu w.\,\{\nu x.\,\{!x(z).\,[\![N]\!]_z \mid \nu u.\,\{!u(z).\,[\![N]\!]_z \mid [\![M]\!]_w[u/x]\}\} \mid w.\mathsf{case}([\![P]\!]_y; [\![Q]\!]_y)\} \\
& = & \nu w.\,\{\nu x.\,\{!x(z).\,[\![N]\!]_z \mid [\![(M[u/x])[N/u]]\!]_w\} \mid w.\mathsf{case}([\![P]\!]_y; [\![Q]\!]_y)\} \\
& = & \nu w.\,\{\nu x.\,\{!x(z).\,[\![N]\!]_z \mid [\![M[N/x]]\!]_w\} \mid w.\mathsf{case}([\![P]\!]_y; [\![Q]\!]_y)\} \\
& \equiv_{assoc} & \nu x.\,\{!x(z).\,[\![N]\!]_z \mid \nu w.\,\{[\![M[N/x]]\!]_w \mid w.\mathsf{case}([\![P]\!]_y; [\![Q]\!]_y)\}\} \\
& \equiv_{swap} & \nu x.\,\{!x(z).\,[\![N]\!]_z \mid \nu w.\,\{w.\mathsf{case}([\![P]\!]_y; [\![Q]\!]_y) \mid [\![M[N/x]]\!]_w\}\} \\
& \equiv_{assoc} & \nu w.\,\{\nu x.\,\{!x(z).\,[\![N]\!]_z \mid w.\mathsf{case}([\![P]\!]_y; [\![Q]\!]_y)\} \mid [\![M[N/x]]\!]_w\} \\
& \equiv_{swap} & \nu w.\,\{\nu x.\,\{w.\mathsf{case}([\![P]\!]_y; [\![Q]\!]_y) \mid !x(z).\,[\![N]\!]_z\} \mid [\![M[N/x]]\!]_w\} \\
& \Longrightarrow_{\kappa_\&} & \nu w.\,\{w.\mathsf{case}(\nu x.\,\{!x(z).\,[\![N]\!]_z \mid [\![P]\!]_y\}; \nu x.\,\{!x(z).\,[\![N]\!]_z \mid [\![Q]\!]_y\}) \mid [\![M[N/x]]\!]_w\} \\
& \equiv_{swap} & \nu w.\,\{w.\mathsf{case}(\nu x.\,\{[\![P]\!]_y \mid !x(z).\,[\![N]\!]_z\}; \nu x.\,\{[\![Q]\!]_y \mid !x(z).\,[\![N]\!]_z\}) \mid [\![M[N/x]]\!]_w\} \\
& = & \nu w.\,\{w.\mathsf{case}([\![P[N/x]]\!]_y; [\![Q[N/x]]\!]_y) \mid [\![M[N/x]]\!]_w\} \\
& \equiv_{swap} & \nu w.\,\{[\![M[N/x]]\!]_w \mid w.\mathsf{case}([\![P[N/x]]\!]_y; [\![Q[N/x]]\!]_y)\} \\
& = & [\![\mathsf{case}(M[N/x]; w.\,P[N/x]; w.\,Q[N/x])]\!]_y
\end{array}$$

Here we first use the assoc rule to switch the order of the cuts, then add the substitution $[\![M]\!]_w[u/x][x/u]$ which doesn't change the meaning of the process, but allows us to use the $\beta_{!C}$ reduction rule to duplicate the process $\nu x.\,\{!x(z).\,[\![N]\!]_z\}$ with $[u/x]$. This gives us one part of the result, namely $[\![(M[u/x])[N/u]]\!]_w = [\![M[N/x]]\!]_w$. We then use a series of assocs and swaps until we have a process which we can use the $\kappa_\&$ rule on to duplicate $\nu x.\,\{!x(z).\,[\![N]\!]_z\}$ over $[\![P]\!]_y$ and $[\![Q]\!]_y$, which after a couple of swaps gives the expected

result.

Case: $[\![\langle P, Q\rangle[N/x]]\!]_y \stackrel{\text{def}}{=} [\![\langle P[N/x], Q[N/x]\rangle]\!]_y$

$$\begin{array}{lll}
[\![\langle P, Q\rangle[N/x]]\!]_y & \stackrel{\text{def}}{=} & \nu x.\,\{!x(z).\,[\![N]\!]_z \mid [\![\langle P, Q\rangle]\!]_y\} \\
& = & \nu x.\,\{!x(z).\,[\![N]\!]_z \mid y.\mathsf{case}([\![P]\!]_y; [\![Q]\!]_y)\} \\
& \equiv_{swap} & \nu x.\,\{y.\mathsf{case}([\![P]\!]_y; [\![Q]\!]_y) \mid !x(z).\,[\![N]\!]_z\} \\
& \Longrightarrow_{\kappa_\&} & y.\mathsf{case}(\nu x.\,\{[\![P]\!]_y \mid !x(z).\,[\![N]\!]_z\}; [\![Q]\!]_y \mid !x(z).\,[\![N]\!]_z) \\
& \equiv_{swap} & y.\mathsf{case}(\nu x.\,\{!x(z).\,[\![N]\!]_z \mid [\![P]\!]_y\}; \nu x.\,\{!x(z).\,[\![N]\!]_z \mid [\![Q]\!]_y\}) \\
& = & y.\mathsf{case}([\![P[N/x]]\!]_y; [\![Q[N/x]]\!]_y) \\
& = & [\![\langle P[N/x], Q[N/x]\rangle]\!]_y
\end{array}$$

Here, we simply use $\kappa_\&$ surrounded by two swaps to duplicate the substitution formula $\nu x.\,\{!x(z).\,[\![N]\!]_z\}$ over both subprocesses $[\![P]\!]_y$ and $[\![Q]\!]_y$ to achieve the result. We can do this because the case statement communicates along channel y, but the cut communicates on channel x so there is no inteference.

Case: $[\![\pi_1(M)[N/x]]\!]_y \stackrel{\text{def}}{=} [\![\pi_1(M[N/x])]\!]_y$

$$\begin{array}{lll}
[\![\pi_1(M)[N/x]]\!]_y & \stackrel{\text{def}}{=} & \nu x.\,\{!x(z).\,[\![N]\!]_z \mid [\![\pi_1(M)]\!]_y\} \\
& = & \nu x.\,\{!x(z).\,[\![N]\!]_z \mid \nu w.\,\{w[\mathsf{inl}].\,w \leftrightarrow y \mid [\![M]\!]_w\}\} \\
& \equiv_{swap} & \nu x.\,\{!x(z).\,[\![N]\!]_z \mid \nu w.\,\{[\![M]\!]_w \mid w[\mathsf{inl}].\,w \leftrightarrow y\}\} \\
& \equiv_{assoc} & \nu w.\,\{\nu x.\,\{!x(z).\,[\![N]\!]_z \mid [\![M]\!]_w\} \mid w[\mathsf{inl}].\,w \leftrightarrow y\} \\
& \equiv_{swap} & \nu w.\,\{w[\mathsf{inl}].\,w \leftrightarrow y \mid \nu x.\,\{!x(z).\,[\![N]\!]_z \mid [\![M]\!]_w\}\} \\
& = & \nu w.\,\{w[\mathsf{inl}].\,w \leftrightarrow y \mid [\![M[N/x]]\!]_w\} \\
& = & [\![\pi_1(M[N/x])]\!]_y
\end{array}$$

For this case, we simply rearrange the cuts and subprocesses with two swaps and an assoc to achieve the desired result. The case for $\pi_2(M)[N/x]$ is analogous.

Case: $[\![(\lambda w.\,M)[N/x]]\!]_y \stackrel{\text{def}}{=} [\![\lambda w.\,M[N/x]]\!]_y$

$$\begin{array}{lll}
[\![(\lambda w.\,M)[N/x]]\!]_y & \stackrel{\text{def}}{=} & \nu x.\,\{!x(z).\,[\![N]\!]_z \mid [\![\lambda w.\,M]\!]_y\} \\
& = & \nu x.\,\{!x(z).\,[\![N]\!]_z \mid y(w).\,[\![M]\!]_y\} \\
& \equiv_{swap} & \nu x.\,\{y(w).\,[\![M]\!]_y \mid !x(z).\,[\![N]\!]_z\} \\
& \Longrightarrow_{\kappa_\parr} & y(w).\,\nu x.\,\{[\![M]\!]_y \mid !x(z).\,[\![N]\!]_z\} \\
& \equiv_{swap} & y(w).\,\nu x.\,\{!x(z).\,[\![N]\!]_z \mid [\![M]\!]_y\} \\
& = & y(w).\,[\![M[N/x]]\!]_y \\
& = & [\![\lambda w.\,M[N/x]]\!]_y
\end{array}$$

Here, we use two swaps surrounding $\kappa_\parr$ to pull $y(w).$ out of the cut as the cut communicates along x but the input communicates along y. This gives us the required result.

Case: $[\![(P(Q))[N/x]]\!]_y \stackrel{\text{def}}{=} [\![(P[N/x](Q[N/x]))]\!]_y$

$$
\begin{array}{lll}
[\![(P(Q))[N/x]]\!]_y & \stackrel{\text{def}}{=} & \nu x.\{!x(z).[\![N]\!]_z \mid [\![P(Q)]\!]_y\} \\
& = & \nu x.\{!x(z).[\![N]\!]_z \mid \nu\alpha.\{\nu\sigma.\{\sigma[w].(w \leftrightarrow \alpha \mid y \leftrightarrow \sigma) \mid [\![P]\!]_\sigma\} \mid !\alpha(u).[\![Q]\!]_u\}\} \\
& \equiv_{assoc} & \nu\alpha.\{\nu x.\{!x(z).[\![N]\!]_z \mid \nu\sigma.\{\sigma[w].(w \leftrightarrow \alpha \mid y \leftrightarrow \sigma) \mid [\![P]\!]_\sigma\}\} \mid !\alpha(u).[\![Q]\!]_u\} \\
& = & \nu\alpha.\{\nu x.\{!x(z).[\![N]\!]_z \mid \nu\sigma.\{\sigma[w].(w \leftrightarrow \alpha \mid y \leftrightarrow \sigma) \mid [\![P]\!]_\sigma\}[\tau/x][x/\tau]\} \mid !\alpha(u).[\![Q]\!]_u\} \\
\\
& \Longrightarrow_{\beta_{!C}} & \nu\alpha.\{\nu x.\{!x(z).[\![N]\!]_z \mid \nu\tau.\{!\tau(z).[\![N]\!]_z \mid \nu\sigma.\{\sigma[w].(w \leftrightarrow \alpha \mid y \leftrightarrow \sigma) \mid [\![P]\!]_\sigma\}[\tau/x]\}\} \mid !\alpha(u).[\![Q]\!]_u\} \\
& = & \nu\alpha.\{\nu x.\{!x(z).[\![N]\!]_z \mid \nu\tau.\{!\tau(z).[\![N]\!]_z \mid \nu\sigma.\{\sigma[w].(w \leftrightarrow \alpha \mid y \leftrightarrow \sigma)[\tau/x] \mid [\![P]\!]_\sigma[\tau/x]\}\}\} \mid !\alpha(u).[\![Q]\!]_u\} \\
& \equiv_{swap} & \nu\alpha.\{\nu x.\{!x(z).[\![N]\!]_z \mid \nu\tau.\{!\tau(z).[\![N]\!]_z \mid \nu\sigma.\{[\![P]\!]_\sigma[\tau/x] \mid \sigma[w].(w \leftrightarrow \alpha \mid y \leftrightarrow \sigma)\}\}\} \mid !\alpha(u).[\![Q]\!]_u\} \\
& \equiv_{assoc} & \nu\alpha.\{\nu x.\{!x(z).[\![N]\!]_z \mid \nu\sigma.\{\nu\tau.\{!\tau(z).[\![N]\!]_z \mid [\![P]\!]_\sigma[\tau/x]\} \mid \sigma[w].(w \leftrightarrow \alpha \mid y \leftrightarrow \sigma)\} \mid !\alpha(u).[\![Q]\!]_u\} \\
\\
& = & \nu\alpha.\{\nu x.\{!x(z).[\![N]\!]_z \mid \nu\sigma.\{[\![(P[\tau/x])[N/\tau]]\!]_\sigma \mid \sigma[w].(w \leftrightarrow \alpha \mid y \leftrightarrow \sigma)\} \mid !\alpha(u).[\![Q]\!]_u\}\} \\
& = & \nu\alpha.\{\nu x.\{!x(z).[\![N]\!]_z \mid \nu\sigma.\{[\![P[N/x]]\!]_\sigma \mid \sigma[w].(w \leftrightarrow \alpha \mid y \leftrightarrow \sigma)\} \mid !\alpha(u).[\![Q]\!]_u\}\} \\
& \equiv_{assoc} & \nu x.\{!x(z).[\![N]\!]_z \mid \nu\alpha.\{\nu\sigma.\{[\![P[N/x]]\!]_\sigma \mid \sigma[w].(w \leftrightarrow \alpha \mid y \leftrightarrow \sigma)\} \mid !\alpha(u).[\![Q]\!]_u\}\} \\
& \equiv_{swap} & \nu x.\{!x(z).[\![N]\!]_z \mid \nu\alpha.\{!\alpha(u).[\![Q]\!]_u \mid \nu\sigma.\{[\![P[N/x]]\!]_\sigma \mid \sigma[w].(w \leftrightarrow \alpha \mid y \leftrightarrow \sigma)\}\}\} \\
\\
& \equiv_{assoc} & \nu\alpha.\{\nu x.\{!x(z).[\![N]\!]_z \mid !\alpha(u).[\![Q]\!]_u\} \mid \nu\sigma.\{[\![P[N/x]]\!]_\sigma \mid \sigma[w].(w \leftrightarrow \alpha \mid y \leftrightarrow \sigma)\}\} \\
& \Longrightarrow_{\kappa_!} & \nu\alpha.\{!\alpha(u).\nu x.\{!x(z).[\![N]\!]_z \mid [\![Q]\!]_u\} \mid \nu\sigma.\{[\![P[N/x]]\!]_\sigma \mid \sigma[w].(w \leftrightarrow \alpha \mid y \leftrightarrow \sigma)\}\} \\
& = & \nu\alpha.\{!\alpha(u).[\![Q[N/x]]\!]_u \mid \nu\sigma.\{[\![P[N/x]]\!]_\sigma \mid \sigma[w].(w \leftrightarrow \alpha \mid y \leftrightarrow \sigma)\}\} \\
& \equiv_{swap} & \nu\alpha.\{\nu\sigma.\{[\![P[N/x]]\!]_\sigma \mid \sigma[w].(w \leftrightarrow \alpha \mid y \leftrightarrow \sigma)\} \mid !\alpha(u).[\![Q[N/x]]\!]_u\} \\
\\
& \equiv_{swap} & \nu\alpha.\{\nu\sigma.\{\sigma[w].(w \leftrightarrow \alpha \mid y \leftrightarrow \sigma) \mid [\![P[N/x]]\!]_\sigma\} \mid !\alpha(u).[\![Q[N/x]]\!]_u\} \\
& = & [\![(P[N/x](Q[N/x]))]\!]_y
\end{array}
$$

Finally, for the application case we begin by switching the order of the cuts with assoc, then we use the same trick that we used for case. We add two substitutions which don't change the meaning of the process, but allow us to duplicate the substitution formula using the $\beta_{!C}$ reduction rule. We then do a series of assocs and swaps before using the $\kappa_!$ rule to pull $!\alpha(u).$ out of the cut on x. Finally we do two more swaps and achieve the desired result.

4.1.2 Reduction Simulation

Now we have proved that we have a valid substitution lemma, we can begin our proof of simulation to show that our translation is indeed correct.

$$
\begin{array}{lll}
\left[\!\!\left[\dfrac{\text{S-Case-In-L}}{\mathsf{case}(\mathsf{inl}(M); y.P; y.Q) \longmapsto P[M/y]}\right]\!\!\right]_x & \stackrel{\text{def}}{=} & \dfrac{}{[\![\mathsf{case}(\mathsf{inl}(M); y.P; y.Q)]\!]_x \longmapsto [\![P[M/y]]\!]_x} \\
\\
[\![\mathsf{case}(\mathsf{inl}(M); y.P; y.Q)]\!]_x & \stackrel{\text{def}}{=} & \nu y.\{[\![\mathsf{inl}(M)]\!]_y \mid y.\mathsf{case}([\![P]\!]_x; [\![Q]\!]_x)\} \\
& = & \nu y.\{y[\mathsf{inl}].!y(z).[\![M]\!]_z \mid y.\mathsf{case}([\![P]\!]_x; [\![Q]\!]_x)\} \\
& \Longrightarrow_{\beta_{\oplus\&}} & \nu y.\{!y(z).[\![M]\!]_z \mid [\![P]\!]_x\} \\
& = & [\![P[M/y]]\!]_x
\end{array}
$$

For S-Case-In-L we simply extract the translation of the case term then use the $\beta_{\oplus\&}$ cut reduction rule to handle the $\nu y.\{y[\mathsf{inl}]. \mid y.\mathsf{case}(;)\}$ and achieve the translation of the result. The proof for case-in-r is analogous.

$$\left[\!\!\left[\dfrac{M \longmapsto M'}{\mathsf{case}(M; y.\,P; y.\,Q) \longmapsto \mathsf{case}(M'; y.\,P; y.\,Q)}\right]\!\!\right]_x \stackrel{\text{def}}{=} \dfrac{[\![M]\!]_y \longmapsto [\![M']\!]_y}{[\![\mathsf{case}(M; y.\,P; y.\,Q)]\!]_x \longmapsto [\![\mathsf{case}(M'; y.\,P; y.\,Q)]\!]_x}$$

$$\begin{aligned}
[\![\mathsf{case}(M; y.\,P; y.\,Q)]\!]_x &\stackrel{\text{def}}{=} \nu y.\,\{[\![M]\!]_y \mid y.\mathsf{case}([\![P]\!]_x; [\![Q]\!]_x)\} \\
&\Longrightarrow_{SCase} \nu y.\,\{[\![M']\!]_y \mid y.\mathsf{case}([\![P]\!]_x; [\![Q]\!]_x)\} \\
&= [\![\mathsf{case}(M'; y.\,P; y.\,Q)]\!]_x
\end{aligned}$$

We know that $[\![M]\!]_y \longmapsto [\![M']\!]_y$ by our induction hypothesis, so applying the S-Case rule we obtain the required result.

$$\left[\!\!\left[\dfrac{}{\pi_1(\langle M, N\rangle) \longmapsto M}\right]\!\!\right]_x \stackrel{\text{def}}{=} \dfrac{}{[\![\pi_1(\langle M, N\rangle)]\!]_x \longmapsto [\![M]\!]_x}$$

$$\begin{aligned}
[\![\pi_1(\langle M, N\rangle)]\!]_x &\stackrel{\text{def}}{=} \nu y.\,\{y[\mathsf{inl}].\, y \leftrightarrow x \mid [\![\langle M, N\rangle]\!]_y\} \\
&= \nu y.\,\{y[\mathsf{inl}].\, y \leftrightarrow x \mid y.\mathsf{case}([\![M]\!]_y; [\![N]\!]_y)\} \\
&\Longrightarrow_{\beta_{\oplus\&}} \nu y.\,\{y \leftrightarrow x \mid [\![M]\!]_y\} \\
&\Longrightarrow_{axCut} [\![M]\!]_y[x/y] \\
&= [\![M]\!]_x
\end{aligned}$$

For S-Prj-Pair-l we find the translation of the term then use the $\beta_{\oplus\&}$ cut reduction rule again to handle the $\nu y.\,\{y[\mathsf{inl}].\mid y.\mathsf{case}(;)\}$, then we use the AxCut rule to substitute x for y in $[\![M]\!]_y$. This results in the translation of M with respect to x which is what we needed. Again, the proof for prj-pair-r will be much the same.

$$\left[\!\!\left[\dfrac{M \longmapsto M'}{\pi_1(M) \longmapsto \pi_1(M')}\right]\!\!\right]_x \stackrel{\text{def}}{=} \dfrac{[\![M]\!]_y \longmapsto [\![M']\!]_y}{[\![\pi_1(M)]\!]_x \longmapsto [\![\pi_1(M')]\!]_x}$$

$$\begin{aligned}
[\![\pi_1(M)]\!]_x &\stackrel{\text{def}}{=} \nu y.\,\{y[\mathsf{inl}].\, y \leftrightarrow x \mid [\![M]\!]_y\} \\
&\Longrightarrow_{SPrjL} \nu y.\,\{y[\mathsf{inl}].\, y \leftrightarrow x \mid [\![M']\!]_y\} \\
&= [\![\pi_1(M')]\!]_x
\end{aligned}$$

We know by our induction hypothesis that $[\![M]\!]_y \longmapsto [\![M']\!]_y$, so applying the S-Prj-l rule, we achieve the necessary result. The proof for S-Prj-r is analogous.

$$
\begin{aligned}
\left[\!\!\left[\dfrac{\text{S-Beta}}{(\lambda x.\, M)(N) \longmapsto M[N/x]}\right]\!\!\right]_y &\overset{\text{def}}{=} && \overline{[\![(\lambda x.\, M)(N)]\!]_y \longmapsto [\![M[N/x]]\!]_y} \\
[\![(\lambda x.\, M)(N)]\!]_y &\overset{\text{def}}{=} && \nu x.\, \{xy.\mathsf{depar}\{[\![\lambda x.M]\!]_z\} \mid !x(u).\, [\![N]\!]_u\} \\
&= && \nu x.\, \{\nu z.\, \{z[w].\, (w \leftrightarrow x \mid y \leftrightarrow z) \mid (z(x).\, [\![M]\!]_z)\} \mid !x(u).\, [\![N]\!]_u\} \\
&= && \nu x.\, \{\nu z.\, \{z[w].\, (w \leftrightarrow x \mid y \leftrightarrow z) \mid (z(x).\, [\![M]\!]_z)[w/x]\} \mid !x(u).\, [\![N]\!]_u\} \\
&= && \nu x.\, \{\nu z.\, \{z[w].\, (w \leftrightarrow x \mid y \leftrightarrow z) \mid (z(w).\, [\![M]\!]_z[w/x])\} \mid !x(u).\, [\![N]\!]_u\} \\
&\Longrightarrow_{\beta_{\otimes\parr}} && \nu x.\, \{\nu w.\, \{w \leftrightarrow x \mid \nu z.\, \{y \leftrightarrow z \mid [\![M]\!]_z\}\} \mid !x(u).\, [\![N]\!]_u\} \\
&\Longrightarrow_{axCut} && \nu x.\, \{\nu w.\, \{w \leftrightarrow x \mid [\![M]\!]_y\} \mid !x(u).\, [\![N]\!]_u\} \\
&\Longrightarrow_{axCut} && \nu x.\, \{[\![M]\!]_y \mid !x(u).\, [\![N]\!]_u\} \\
&\equiv_{swap} && \nu x.\, \{!x(u).\, [\![N]\!]_u \mid [\![M]\!]_y\} \\
&= && [\![M[N/x]]\!]_y
\end{aligned}
$$

For S-Beta we begin as usual by extracting the translation of the term. Then we substitute w for x in $z(w).\, [\![M]\!]_z$ so we can use the $\beta_{\otimes\parr}$ cut reduction rule to handle $\nu z.\, \{z[w].\ \mid z(w).\, \}$. Then we use the axCut rule to substitute y for z in $[\![M]\!]_z$ resulting in $[\![M]\!]_y$. We then use axCut again to substitue x for w in $[\![M]\!]_y$ which changes nothing. Finally we use the swap equivalence to switch the positions of the subterms in $\nu x.\, \{\dots\}$ to achieve the translation of the result as required.

Chapter 5

Conclusion

In this thesis we have explored classical linear logic as a session typed process calculus, and how this relates to concurrency. We have given a provably correct call-by-name translation from the λ-calculus into classical processes and also proved substitution for this translation. This shows us how sequential languages can be translated into concurrent languages and how translation impacts the processes being translated.

There are, of course, two translations we could have considered; call-by-name, and call-by-value. We chose to consider call-by-name since this is the standard used for the λ-calculus. However, it would be interesting to see how a call-by-value translation would differ from the one presented here. In Girard's original work on classical linear logic, he offered two separate translations from intuitionistic logic into linear logic. The first is the one used to guide the translation presented in this thesis. The second translation differs from the first and would give the following if we chose to do the translation from λ-calculus into classical linear logic:

$$\begin{aligned}(A + B)^* &= A^* \oplus B^* \\ (A \times B)^* &= A^* \otimes B^* \\ (A \to B)^* &= !({A^*}^\perp \parr B^*)\end{aligned}$$

Although Girard dismissed this as a boring translation, Toninho et al. [9] used it to create a call-by-value translation from the λ-calculus into the π-calculus. It would be interesting to follow along these lines to see how a call-by-value translation from the λ-calculus into CP would look.

5.1 Future Work

This thesis provides the logical foundations for a message-passing concurrent programming language which uses session types to ensure type safety. Currently, the only concurrent language (that we know of) which has its basis in linear logic and session types is Concurrent C0 [1] which is used as a teaching language for students learning concurrent programming. We could create a full-scale language based on these ideas using the logic included here and practical implementation of C0.

We can also consider what happens when we have $\otimes = \parr$ in CLL and what we need to add to CLL in order to translate untyped π-calculus into CLL with these unknown additions.

Bibliography

[1] Rob Arnold. "C0, an imperative programming language for novice computer scientists". PhD thesis. Master's thesis, Department of Computer Science, Carnegie Mellon University, 2010.

[2] Luís Caires and Frank Pfenning. "Session types as intuitionistic linear propositions". In: *International Conference on Concurrency Theory*. Springer. 2010, pp. 222–236.

[3] A Church. "A set of postulates for the foundations of mathematics". In: *Ann. of Math.(2)* 33 (1932), pp. 346–366.

[4] Alonzo Church. "A formulation of the simple theory of types". In: *The journal of symbolic logic* 5.2 (1940), pp. 56–68.

[5] Jean-Yves Girard. "Linear logic". In: *Theoretical Computer Science* 50.1 (1987), pp. 1–101. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(87)90045-4.

[6] Stephen C Kleene and J Barkley Rosser. "The inconsistency of certain formal logics". In: *Annals of Mathematics* (1935), pp. 630–636.

[7] Wen Kokke, Fabrizio Montesi, and Marco Peressotti. "Taking linear logic apart". In: *arXiv preprint arXiv:1904.06848* (2019).

[8] Robin Milner, Joachim Parrow, and David Walker. "A calculus of mobile processes, i". In: *Information and computation* 100.1 (1992), pp. 1–40.

[9] Bernardo Toninho, Luís Caires, and Frank Pfenning. "Functions as session-typed processes". In: *International Conference on Foundations of Software Science and Computational Structures*. Springer. 2012, pp. 346–360.

[10] Philip Wadler. "Propositions as sessions". In: *Journal of Functional Programming* 24.2-3 (2014), pp. 384–418.