The State of Process Calculi and Concurrency

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Andrew J. Roetker

May 2014

Approved for the Division
(Mathematics)

James D. Fix

Acknowledgements

There were a lot of people who made Reed, and thus this thesis, possible for me. I would like to thank all the dormies who make campus life fun, all the professors who made coming to class a delight, and the friends who made life more interesting.

To Jim: Thanks for all the distractions, I've learned so much from you!

To Pfenning: Thanks for the C-H Isomorphism!

To those special Weisensee's: Thanks for the pinochle.

To Will: Thanks for the variety hangouts and salmon.

To Cauchy: You are a pain in the ass and I love you.

To Mom and Dad: Thanks for everything.

To Hambone: Ees okai, I will ALWAYS get your earplugs.

Table of Contents

Abstract

This thesis investigates a Curry-Howard correspondence between linear logic and a type system for a typed synchronous π-calculus. We give a specification for a process calculus and demonstrate how to reason about its concurrent programs. We also provide an introduction to intuitionistic linear logic, a logic of resources, providing explanations and examples of its connectives. The session-typed process terms and logical rules will be in one-to-one correspondence. This correspondence provides information about programs written in our concurrent programming language.

Chapter 1

Introduction

1.1 Internet infrastructure and modern programming languages

My friend messaged me the other day. He wanted to know if Haskell was a good language to learn to make writing the *network code* he was working on easier. At the time his work used a dialect of PHP called Hack which has a type system.

> i like that when i'm developing in hack i can write code then run "hh_client" (which does the type checking) and then eliminate like 90% of the bugs i introduced in my code
>
> and no side effects could theoretically make that even better
>
> — Tim Bauman

In essence he was interested in saving himself time programming by allowing the type checker to debug his code for him. To its proponents the functional style of programming and the type system in Haskell provide a framework for code expression and code checking that make certain code development less prone to error. Unfortunately network application code is one of the "awkward squad" in Haskell (Jones, 2001) — not easy to express — and its type system is not rich enough to provide certain guarantees about its correctness.

Computing infrastructure is complex. The Cloud is made up of many components which communicate with each other. We often view the way the components are constructed as being made up of clients and servers, that is, we might identify some component as providing a particular network service, the server, and another component which use that service, its client. The server could be the Amazon website and the notion of a client could be you as a user browsing on your phone, tablet and laptop to construct your Amazon shopping cart. The client and server software has typically been written in a *C*-like language, running on operating systems with *Unix*-like components. Clients make TCP/IP connections to a server using a network *API* (Application Programmers' Interface) similar to Berkeley sockets. A socket connection allows for the back and forth communication of information between two

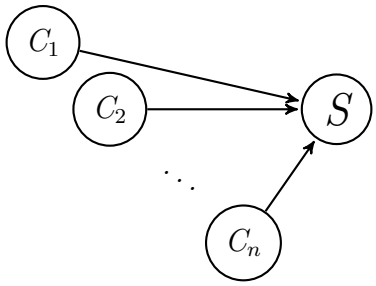

Figure 1.1: n clients to 1 server architecture

programs running on the same or different computers. Normally, in order to handle numerous client requests, the server is multithreaded using an API similar to POSIX threads. These threads run concurrently, and are constructed as independently executing processing agents, even on a computer with a single processor.

Today's software systems are distributed amongst many computers and, within any computer, run as a system of concurrent processes.

Furthermore code itself is organized into these components. Software may link several components in a single piece of code. A server may actually offer up HTTP files using a templating API which accesses a database back-end using another API.

Building robust distributed systems made up of numerous components is notoriously difficult. Concurrent programming is prone to being buggy. Because there could be simultaneous access to shared data or services there may be race conditions and there could be deadlock. These kinds of bugs don't always manifest themselves, making them hard to reproduce and difficult to pin down. Engineers like my friend Tim would like some assurance that the code he is writing is correct and that the system specification is safe.

Software firms have testing departments whose sole job consists of writing code to test the code developers are working on. Some modern programming systems have facilities that make testing easier and also allow for the handling of errors as they arise. Programmers follow careful disciplines to prevent shipping faulty code, for example writing tests for code even before they have written the code itself.

One way to gain some assurances is to augment a programming system to include components for formal specification and verification of code. Some researchers consider mechanisms for checking aspects of developed software. A simple example is compiler technology which checks for type safety and correctness of its programs.

Design of new programming languages is often driven by these concerns. This has led to a proliferation of programming languages each one addressing some aspect of code development. Today there are a number of internet languages such as GoLang, Javascript, Clojure, Java, Ruby and Python (whose influence relationship is depicted in figure 1.2). Some of these languages even address concurrency issues by defining useful "concurrency primitives". Figure 1.2 lays out some of the programming languages and programming language models used for the internet infrastructure and mentioned in this thesis. Many of the languages share features including how programmers define functions, objects, and express concurrency and

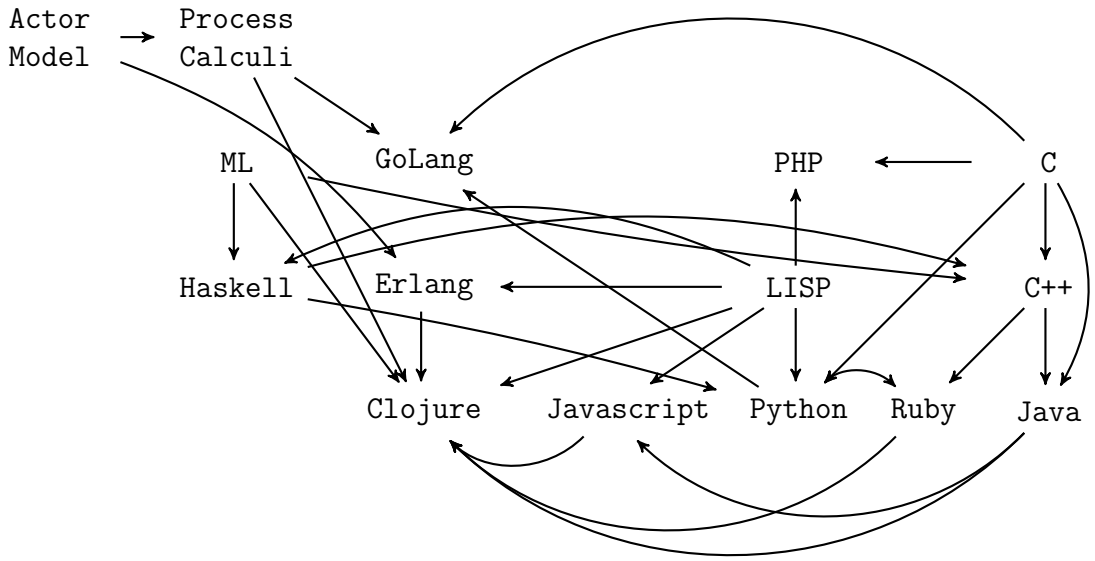

Figure 1.2: Programming Languages of this thesis with an influence relationship from Wikipedia (2014)

networking. The arrows illustrate their influence relationship as per their Wikipedia page entries. Some have concurrency primitives such as channels/processes, threading, or actors/messaging built into the language itself while most have support for concurrency in popular libraries for the language.

1.2 Functional programming languages

In another spectrum of the programming language world are the functional programming languages (FPLs), such as ML (Milner et al., 1997), Lisp (McCarthy, 1960), and the pure functional programming language Haskell (Hudak et al., 2007). Many of the internet programming languages we described use FPL principles including immutable data types, functions as first-class values, and functional design patterns. One of the key features of ML and Haskell is their Hindley/Milner type system.

This system performs simple checks on its computer language's code—making sure you didn't get your *int list* in my *char* var(iable). It also provides an inference system that reasons about parametrized types. To get a sense of what these are consider Java Generic types, which were developed by Wadler by adapting Haskell's type system to Java (Wadler, 2006).

FPLs are based on the λ-calculus (Church, 1985) (in some sense the "purest" functional language), which has a "black box" or "batch" processing view of computing. Programs receive input, compute, and output a value. There are no side-effects in the λ-calculus, no stateful world the functions interact with. Discovery of Hindley/Milner type systems came from the investigation of the typed λ-calculus. The typed λ-calculus has a Curry-Howard isomorphism (Curry, 1934) with propositional

logic, where proofs correspond to programs and propositions to types. This is a powerful correspondence with deep implications for programming languages.

> [We trace] one concept, second-order quantification, from its inception in the symbolic logic of Frege through to the generic features introduced in Java 5 ... The remarkable correspondence between natural deduction and functional programming informed the design of type classes in Haskell. Generics in Java evolved directly from Haskell type classes, and are designed to support evolution from legacy code to generic code.
>
> —Wadler (2006)

In short, FPLs make it apparent that there is a close link between systems of logic, deduction and inference, and reasoning about computer programs.

1.3 A Concurrent Type Theory

Milner et al. (1992) attempted to develop a calculus for specifying "realistic" computer systems, seeking to characterize them as a collection of concurrent and communicating processes that interact. The outcome of that research was a process calculi known as CCS (Calculus for Communicating Systems). The refinement of CCS led to the creation of the π-calculus which has been used as a model for mobile systems by some authors (Milner et al., 1992). As yet these calculi have not caused a big uproar in the developer community, though some have been used as a basis for language design. For example CSP (Hoare, 1985) (Communicating Sequential Processes) with a channel-based concurrency model has influenced concurrency mechanisms in *GoLang* (Pike & Gerrand, 2013) and *Clojure* (Hickey, 2013).

The natural question that arises for programming language researchers is whether there is a Curry-Howard isomorphism between a process calculus and a logic. This thesis investigates a fruitful correspondence between linear logic and a type system for a typed synchronous π-calculus. In Chapter 1 we give a specification for our process calculus showing how one creates an abstract programming language for reasoning about concurrent systems. In Chapter 2 we give a brief introduction to linear logic and its connectives. We will demonstrate how linear logic is a logic of resources and stateful reasoning. Chapter 3 unites its two predecessors by giving a Curry-Howard correspondence between linear logic and the session types for π-calculus process terms This correspondence provides information about programs written in our concurrent programming language.

Chapter 2

A Language for Concurrent Computation

2.1 Motivation

The Math Department has hired a new executive assistant, Cathy. Cathy is excited to start helping all the teachers and students, but needs to know what to do to start working. Cathy's duties include directing phone calls to the intended professor in the department, signing off on timesheets from graders and passing them on to the business office, and scheduling thesis orals. Is there a natural way to tell Cathy how to work? To describe Cathy herself?

Below we lay out some simple pseudo-code for some object-oriented language which implements the Cathy work loop. Cathy works prepared to handle interruptions for any of the tasks for which she is assigned. For each of these tasks she has a *method* or procedure for handling that interruption. For example the work loop could be described as follows:

```
Cathy.BeginWork( office ):
  Cathy.Busy = false
  Cathy.Working = true
  while Cathy.Working do
    if not Cathy.Busy then
      if office.PhoneRinging?() then
        Cathy.Busy = true
        Cathy.HandleCall(office.Phone)
        Cathy.Busy = false
      else if office.EmailNotification?() then
        Cathy.Busy = true
        Cathy.HandleEmail(office.Computer.Email)
        Cathy.Busy = false
      else if office.DropInVisitor?() then
        Cathy.Busy = true
```

```
        Cathy.HandlePerson(office.NextInLine())
        Cathy.Busy = false
      end if
    end if
  end while
```

We could implement one of the methods for handling an interruption with a method the inputs a *phone* and uses the *phone* to get the *call*, upon which Cathy can communicate with whomever called. Cathy asks who the call is for and uses the *answer* to connect the person to the desired recipient. Once the call is through Cathy must hang up the connection.

```
Cathy.HandleCall( phone ):
  call := Cathy.AnswerPhone(phone)
  answer := Cathy.AskForWho?(call)
  while answer == "silence" do
    answer = Cathy.AskAgain?(call)
    if Cathy.WaitedTooLong?() then
      Cathy.HangUp()
    end if
  end while
  prof := answer
  if Cathy.Available?(prof) then
    Cathy.ForwardCall(call, prof)
  else
    Cathy.Respond(call, "unavailable")
  end if
  Cathy.HangUp()
```

If the procedure above looks verbose and complicated for answering a phone call, that's because it is. Do we really need to tell Cathy how long to wait for an answer on the phone? Or for that matter should we even need to tell Cathy to wait for an answer at all; isn't waiting for a response an intrinsic part of communication?

Cathy's overall job description is that of a request handler, she processes requests by answering questions or handing off the request to a professor say. Cathy may receive calls over the phone, receive a timesheet online, or have a student drop-in and request for thesis orals so she must be prepared for all possibilities.

We propose that a specification for Cathy should reflect the innate structure of communication. We claim that there is a natural language for expressing the structure and computation of communication, which still captures the necessary expressiveness of the imperative language. Let Cathy be a process which offers services along different interfaces on which we communicate with Cathy. These interfaces in this case might be the office, phone, and computer.

Cathy offers a choice of services to her clients, the students and professors. A post-

it note appears in front of Cathy, and written on it is one of three labels: phoneCall, emailNotification, and dropinvistor. These instruct her which office component should be used next to handle the subsequent request. We view the office as a component of Cathy's job that she may use to receive other components, such as a phone or computer, or send components such as candy to a drop-in visitor. For instance the new notation for describing what Cathy does if she receives a phone call would be

```
Cathy ← office := {                                  // Cathy has access to the office component
office.CASE(                                         // Wait for a tag on the office channel
        phoneRing  ⇒  { office(phone);               // Had phoneRing so pick up the office phone
                        phone(call);                 // Use phone to answer call
                        call⟨Questions⟩;             // Over the call send Questions
                        call(Answers);               // Then wait for the Answers
                        ...;                         // Redirect call, etc.
                        call.WAIT();                 // Here she waits for "Goodbye!"
                        Cathy }                      // Go back to waiting for a tag
  emailNotification  ⇒  {...}
      dropinvisitor  ⇒  {...} )}
```

Where office(phone) is the syntax for receiving a phone over the office channel, similarly a call c received over the phone channel would be phone(c), or the receipt of an email e on the computer computer would be computer(e). Also note that using this language we can very easily see what protocol a user interacting with the Cathy process must follow,

```
User ← office := {  office.phoneRing;
                    office⟨phone⟩;
                    phone⟨call⟩;
                    call(Questions);
                    call⟨RESPONSESTO(Questions)⟩;
                    ...;
                    call.CLOSE⟨⟩;   // Here they say "Goodbye!"
                    User }
```

Request handlers and routers are pervasive in distributed programming, from web-servers which offer to serve static HTML files and post comments or pictures, to query engines which must hand off requests for information to database back-ends which also have a specification for interacting with requests.

The process model is meant to give a language to describe what happens during some communication without needing to give a procedure for exactly how to execute that communication. This model means to do the same thing for concurrent programming as what functional programming intends to do for procedural languages, describe what computations need to take place, not exactly how to implement them.

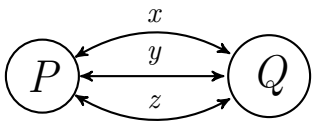

Figure 2.1: Processes P and Q communicate over channel x, y, and z

2.2 Process Model

In the functional setting of the λ-calculus, a function term $\lambda x.M$ can be viewed as a program that takes an input x and does something with it, determined by the term M, producing some result. A particular input may be described by another term N to be "fed" into that program. The λ-calculus prescribes a "reduction mechanism" that gives the result of the application of the term $\lambda x.M$ to the term N. In short, we have a calculus for describing this function application mechanism, a certain family of computations. In the typed λ-calculus, the term you plug in must match the type of input expected as x.

In the concurrent setting of the π-calculus (Milner et al., 1992, e.g.), a process term $z \leftarrow P \leftarrow x_1, x_2, \ldots$ can be viewed as a program which consumes resources provided along the channels $x_1, x_2, \ldots$ and provides a service along z, determined by the term P. A particular resource x_i is provided by another term Q_i. The π-calculus prescribes a "reduction mechanism," analogous to the one for λ-calculus, that gives the result of putting P in parallel with the Q_i terms. Here we have a calculus for describing the communication that proceeds from this composition, another family of computations. In the session-typed π-calculus, the resource provided by some Q_i must match the session type of the resource expected over x_i.

In this section we introduce a process model of computation which integrates a synchronous π-calculus and a functional language. Processes P and Q communicate using channels, which we typically identify with the variable names x, y and z. In the general case of Milner's π-calculus any number of processes may be arbitrarily linked together such as in figure 2.2. In this diagram a is a shared channel between all three processes P, Q, and R. Our presentation restricts the π-calculus model permitting linkages to be shared by two and only two processes in a single state. With this in mind the translation of figure 2.2 to the process model we present might look more like figure 2.3. In fact the process model we present is a synchronous π-calculus and therefore a group of processes which interact with each other must offer a channel with a single set of services. A process may only offer one thing at a time.

These services that a process offers evolve through interaction with other processes. We will annotate the evolution of processes one to another with a labelled transition arrow $P \xrightarrow{\tau} P'$. A process may offer to input or output information along a channel, where information is either basic data, in the form of strings, lists, *etc.*, or a channel to communicate with other processes. These services will allow processes to offer a package of services along a channel. A process may also offer a choice of different services, where the choice may be the user's or the server's. These simple operations allow us to naturally express stateful computation.

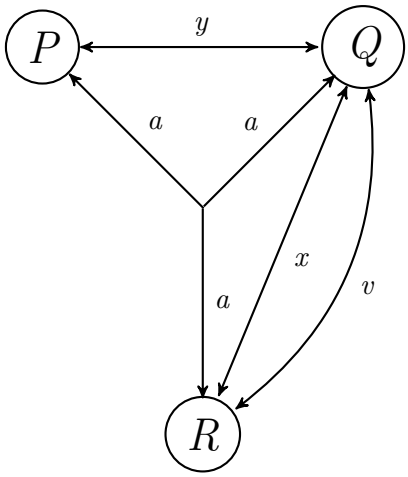

Figure 2.2: Processes P, Q, and R, where all three share a channel a

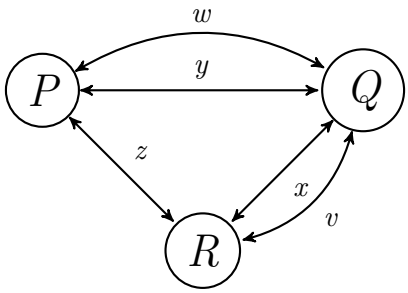

Figure 2.3: Processes P, Q, and R communicate with channels between pairs only

2.2.1 Computation as communication

Processes communicate with neighbors via linkages called channels which are identified by names. In the pure π-calculus computation is represented by communication of channel names across channels, analogous to computation in the pure λ-calculus represented by function application. For instance say two processes P and Q communicate using some channel x. If P offers services along x, and Q uses those services (for which we use the following syntax)

$$x \leftarrow P := \text{“}P \text{ offers services along } x\text{”} \quad \text{and} \quad Q \leftarrow x := \text{“}Q \text{ uses services along } x\text{”}$$

then P and Q may be composed together and the new process expression for this composition is:

$$(\nu x)(P \mid Q)$$

which puts P and Q together in *parallel composition* $(P \mid Q)$, sharing x as a *private channel* indicated by the *name restriction* (νx).

Note that if P provides some service which is compatible with the service which Q provides, if communication can proceed between the two, then we describe their services as *dual* to each other.

2.2.2 Input/Output

Now that we have described how two processes offering compatible services are composed, we describe the kinds of services that processes may offer. The input and output services are fundamental in the π-calculus as they describe the transfer of information between processes.

Suppose we have system in which the process P offers to receive some y over some channel x and Q offers to send some w over x,

$$P := x(y).P' \quad \text{and} \quad Q := x\langle w\rangle.Q'$$

then as long as w offers the service P expects y to provide, the system undergoes an internal transition τ and P now uses w as it would have used y and Q goes on about its business.

$$x(y).P' \mid x\langle w\rangle.Q' \xrightarrow{\tau} P'\{w/y\} \mid Q'$$

The τ transitions are analogous to the β-reductions of the λ-calculus. Note that from the perspective of P the process Q has instantiated a new channel (νw) (ν may be read literally as new), for P to use, e.g. the office instantiates a new call line for Cathy to use. In the pure π-calculus all computation is modeled by this channel passing style, but for practical purposes it is useful to also allow for P and Q to pass basic data-types such as strings and integers to each other, thus the names y and w may refer to either channels or these basic data-types. Context will make the distinction clear when necessary.

Example: Paying the Waiter

To illustrate how input/output works lets consider a common transaction, paying the bill for dinner. The processes are You and the Waiter. Waiter and You communicate using the table component. On the table the Waiter sends you the bill. You use the bill to compute the amount of money to leave plus tip, and withdraw that money from the wallet channel and send that back to the waiter along the table. After that transaction is done the Waiter is a WealthierWaiter and You are a PoorerButSatisfiedYou.

$$\begin{aligned}\text{Waiter} \leftarrow \text{table} := \{ \quad & \text{table}\langle\text{bill}\rangle. \\ & \text{table}(\text{money}).\text{WealthierWaiter} \ \}\end{aligned}$$

The waiter with access to the table channel outputs a bill and inputs money, then continues on as a WealthierWaiter.

$$\begin{aligned}\text{You} \leftarrow \text{table}, \text{wallet} := \{ \quad & \text{table}(\text{bill}). \\ & \text{wallet}\langle\text{bill}\rangle. \\ & \text{wallet}(\text{money}). \\ & \text{table}\langle\text{money}\rangle.\text{PoorerButSatisfiedYou} \ \}\end{aligned}$$

You have access to the wallet and table channels, on the table channel You input the bill, output the bill along your wallet, input the money and output the money along the table, continuing on poorer and satisfied. Composed together, the Waiter and You communicate perfectly and evolve to a better state,

$$(\nu \text{ table})(\text{Waiter} \mid \text{You}) \xrightarrow{\tau} \ldots \xrightarrow{\tau} (\nu \text{ table})(\text{WealthierWaiter} \mid \text{PoorerButSatisfiedYou})$$

The transitions of the composition demonstrate that the process calculus is meant to describe evolving systems. That is to say we want to describe how the state of the Waiter and You evolves over time through a number of transactions.

2.2.3 Synchronous Communication

The process model demonstrated here is a synchronous π-calculus which means that processes must synchronize over a channel on input/output operations before continuing communication. For $x(y).P'$ and $x\langle y\rangle.Q'$ to continue on as P' and Q', P must wait for Q to output along x or Q must wait for P to input along x. This is a reasonable model for a User of Cathy's services. For instance, a User cannot and should not continue with a conversation until Cathy receives the phone call. In other situations though this model is quite restrictive. For example, while Cathy is waiting for a student to answer her questions she may want, to answer an email or two and this model does not allow for this asynchronous communication.

2.2.4 Termination

In the synchronous π-calculus if a process P terminates communication with Q over channel x, P sends null output along x and Q must receive this signal $x()$.

$$x\langle\rangle.P' \mid x().Q' \xrightarrow{\tau} P' \mid Q'$$

This is refered to as a channel termination or handshake. In more concrete syntax we might have x.CLOSE() and x.WAIT(). It is possible to express some in process terms asynchronicity by allowing processes to terminate communication along a channel without this handshake. Think of the difference between the two models in terms of a phone call. In the synchronous model we require that the user on the other end *tell us* that the conversation has ended before we may hang up, but in the asynchronous version we may hang up even if the conversation ends but we weren't *explicitly told* goodbye. For our examples we will assume that process terms which terminate communication along a channel use this handshake to do so.

Incidentally a process which has no open channels to communicate along is referred to as the nullary process $\mathbf{0}$ of the π-calculus, also called the inactive or terminated process. The process $\mathbf{0}$ is the unit of parallel composition in that $P \mid \mathbf{0} \equiv P$.

2.2.5 Identity as Forwarding

With just these tools we may implement a process for the π-calculus which when handed some information i along a channel x, forwards that information along another channel y. This process is so useful we define it as a process primitive for clarity

$$x(i).y\langle i\rangle.\mathbf{0} \equiv [x \leftrightarrow y]$$

2.2.6 Choice and Selection

A process may offer a choice of services, as in the Cathy example where Cathy offers to read emails or answer the phone. For a process P to offer a variety of services along x, P must be prepared for any service Q may select to use.

To communicate which of Cathy's services they wished to use, a student had to give Cathy a post-it note communicating the service. In the π-calculus we refer to the post-it note as a *label*. In general, a client Q sends the label l_i, where i is the index of label, over x to select the service it will use, then will behave as Q'.

$$Q := x.l_i; Q'$$

For P to offer a choice of n services over x, the service offered over x is prepared for any *case* of the label it could recieve.

$$\begin{aligned} P := x.\text{CASE}(\quad & l_1 \Rightarrow P_1, \\ & l_2 \Rightarrow P_2, \\ & \ldots, \\ & l_n \Rightarrow P_n \ ) \end{aligned}$$

If P receives some label l_i over channel x, P will behave as the process P_i. Composing P and Q together we have the following process reduction:

$$Q \mid P \xrightarrow{\tau} Q' \mid P_i$$

This is analogous to the case statement of many imperative languages or the case expression of some functional languages. One may think of these case expressions as syntactic sugar for nested conditionals.

2.2.7 The Big Picture

In the following section we mention some of the unusual restrictions we have placed on the process calculus. As a full treatment of the following is beyond the scope of this thesis, a curious mind can find more details in (Toninho et al., 2014).

In this chapter we introduced a restricted process calculus similar to the one presented in Toninho et al. (2014) which is a combination of a synchronous π-calculus, labeled choice and selection, and basic data terms. The syntax of processes is shown below:

Terms	M, N	$::=$	$M\,N \mid (M, N) \mid \ldots$	(basic data terms)
Processes	P, Q	$::=$	$x\langle M\rangle.P$	term output
		$\mid$	$x\,(y)\,.P$	input
		$\mid$	$(\nu y)x\langle y\rangle.P$	bound output
		$\mid$	$(\nu y)\,.P$	name restriction
		$\mid$	$P \mid Q$	parallel composition
		$\mid$	$x.\textsc{case}\,(\overline{l_j \Rightarrow P_j})$	branching
		$\mid$	$x.l_i; P$	selection
		$\mid$	$[x \leftrightarrow y]$	forwarding
		$\mid$	$\mathbf{0}$	termination

We write $\overline{l_i \Rightarrow P_i} = l_1 \Rightarrow P_1, \ldots, l_n \Rightarrow P_n$ for an indexed case term. The basic data terms are from some well-typed functional language such as the typed λ-calculus are here for convenience. These basic data terms include constructs such as functions, strings, numbers, and lists.

As noted before this process model uses a synchronous π-calculus, but this model further restricts processes by forcing them to provide a single service after the internal interactions have taken place. A process may provide multiple services over a single channel or may continue to interact after providing some service but it may not simultaneously provide services along different channels. This restriction is analogous to the restriction on terms of the λ-calculus to return a single value which allows us to reason and make guarantees about programs. Although this restriction leads to a number a desired behaviors, such as freedom from deadlock and divergent behavior, it removes a lot of parallelism from processes which may *not* be desired. For example even if Cathy is not waiting for a User to answer her questions, i.e. synchronizing on input/output, she cannot be answering an email, no multitasking allowed.

2.2.8 Multiple Input/Outputs

In functional and imperative languages it is useful to implement a pair or list of basic data-types. Similarly it is helpful to describe how P might offer a pair of services to Q and how Q would use this pair. A word of caution though, this is not offering Q a choice between multiple services as with the Cathy example of offering to pay timesheets *or* answer the phone. An example of this is a spa treatment which offers a massage and pedicure services package and you have to use both.

So how does P offer a pair of services together? Clearly P must offer some of the services along a channel y and the other services along a different channel x, but

P may only offer services along a single channel at a time. Consider the following example for a process Spa

$$\begin{aligned} \text{Spa} := (\nu\ \text{pedicurist}) \quad & \text{masseuse}\langle\text{pedicurist}\rangle. \\ & \text{masseuse}\langle\text{massage}\rangle. \\ & \text{pedicurist}\langle\text{pedicure}\rangle.\text{WealthierSpa} \end{aligned}$$

$$\begin{aligned} \text{You} := \quad & \text{masseuse}(\text{pedicurist}). \\ & \text{masseuse}(\text{massage}). \\ & \text{pedicurist}(\text{pedicure}).\text{HappierYou} \end{aligned}$$

Composing the Spa and You together, and after several internal transitions $\overline{\tau_i}$ we have

$$\text{Spa} \mid \text{You} \xrightarrow{\tau} \ldots \xrightarrow{\tau} \quad \text{WealthierSpa} \mid \text{HappierYou}$$

This example demonstrates how a process provides a package of services, first the process binds a new channel which the will provide the first set of services call the channel y, and sends that channel along x, after y has been sent x provides the other set of services in the package.

This example also demonstrates one of the important benefits of this process model. Once a procedure has been made for implementing some service, a programmer implementing the client interface can easily deduce the specification.

2.2.9 Example: P-Fib

To describe computations that are traditionally described with functional computations, many channels must be created and processes spawned which can clutter the syntax of a typically neat looking problem. Here we demonstrate how "cleanly" the π-calculus allows us to express stateful computation with by using a modified classical computer science example.

$$\begin{aligned} z \leftarrow \text{P-Fib}(n) \quad & := \\ & \{ \quad \text{IF } n \leq 1 \text{ THEN} \\ & \qquad z\langle n\rangle.z\langle\rangle \\ & \quad \text{ELSE} \\ & \qquad \{\ x \leftarrow \text{P-Fib}(n-1); \\ & \qquad\ \ y \leftarrow \text{P-Fib}(n-2); \\ & \qquad\ \ x(u).x(); \\ & \qquad\ \ y(v).y(); \\ & \qquad\ \ z\langle u+v\rangle.z\langle\rangle\ \} \quad \} \end{aligned}$$

The process P-Fib is parametrized by some natural number n, and offers a channel z over which it will output an integer and then close the channel. When the process runs, if n is sufficiently small it will output n and close the channel. Otherwise the process will recursively spawn two P-Fib processes and bind the channels they output along to the names x and y. The original process will input along these channels and output their sum along z and then close the channel.

Chapter 3

Linear Logic

> Some of the best things in life are free; and some are not.
>
> Truth is free. Having proved a theorem, you may use this proof as many times as you wish, at no extra cost. Food, on the other hand, has a cost. Having baked a cake, you may eat it only once. If traditional logic is about truth, then linear logic is about food.
>
> —Wadler (1993)

While writing this thesis "Andrew is typing and drinking coffee" was true, but right now this is most likely not true. The change of truth over time is studied in *temporal logic*. In *linear logic* we wish to model how truth changes with the *change of state* of the system, what is true in a pre-Thesis state and in a post-Thesis state. We are therefore concerned with how truth from one state is consumed as a resource to produce new truth.

In this section we introduce *intuitionistic* linear logic originally developed by (Girard, 1995, e.g.) as an "environmentally friendly" logic, that is to say "resource conscious". The model for linear logic is simple, truth is *consumed* when used to *produce* the truth in the conclusion. Assumptions may not be freely copied and all assumptions must be used exactly once. Linear logic is a special case of intuitionistic logic in which we specifically disallow Contraction and Weakening *structural rules*—which is to say resources must be used, from absence of Contraction, and that we cannot freely copy assumptions, from the absence of Weakening.

3.1 Intuitionistic Linear Logic

The linear logic that we discuss here is based on natural deduction. A linear logical *proposition* is defined by the following grammar:

$$A, B := X \mid A \multimap B \mid A \otimes B \mid A \oplus B \mid A \mathbin{\&} B$$

where X ranges over logical constants. This is to say that propositions are built up using the combining forms *linear implication* written $\multimap$ (often read as "lolli"),

multiplicative or *simultaneous conjunction* written $\otimes$ ("and", "tensor"), *disjunction* written $\oplus$ ("or"), and *alternative conjunction* written $\&$ ("with").

We write a *judgment* in the form

$$\underbrace{A_1, \ldots, A_n}_{\Delta} \vdash C$$

meaning that given $A_1, \ldots, A_n \equiv \Delta$ one may conclude or prove C. We refer to Δ as the *resources* or the *assumption* of the judgment which is a sequence of zero or more propositions. We refer to C as the goal. In order to prove C the rules of liner logic require us to use all of Δ once and only once. This form of a judgment is an example of a *sequent* from the *sequent calculus* developed by Gentzen (1935). For example, let pizza, beer, and student be predicates each defined on a subset of the constants cheese, pale ale, ipa, David, and Sally. Given the set of resources pizza(cheese), beer(ipa), and student(x), we may consume these to produce bloated(x), where x is a *schematic variable* that may be instantiated with either Sally or David. Then we write the corresponding judgment as

$$\text{pizza(cheese)},\ \text{beer(ipa)},\ \text{student}(x) \vdash \text{bloated} - \text{student}(x)$$

A *rule of inference* consists of a set of zero or more judgment written above a line and exactly one judgment written below the line and is written as

$$\frac{\Delta_1 \vdash C_1 \ldots \Delta_n \vdash C_n}{\Delta \vdash C}\ \mathsf{label}$$

If all the judgments above the line are derivable then the judgment below the line is also derivable. In words we say that if Δ_1 proves C_1, Δ_2 proves C_2, ..., and Δ_n proves C_n, then if we have Δ, we may use the label rule and consume Δ to prove C. The label is the *label* or *name* of the rule, which is used for rules which are commonly referred to.

Inference rules are concerned with the propositions we define for our logic. We may also have *judgmental rules* or *structural rules*—so called because they are concerned with the nature of judgments, resources and goals, and examine the structure of the sequent not propositions.

3.1.1 Weakening and Contraction

Again the basic idea of linear logic is that we are resource conscious, all propositions in the context Δ, for some judgment $\Delta \vdash C$, must be used *exactly* once to produce C. This is reflected by the absence of the *structural rules weakening* and *contraction/copy*, present in traditional intuitionistic logic, from intuitionistic linear logic,

$$\frac{\Delta \vdash C}{\Delta, A \vdash C}\ \mathsf{weakening} \quad \text{and} \quad \frac{\Delta, A, A \vdash C}{\Delta, A \vdash C}\ \mathsf{contraction/copy}$$

Weakening tells us that if given a *dollar* you can prove a *soda*, we may conclude that given a *dollar and* a *quarter* you can prove a *soda*. This precisely means that we don't need to use resource *quarter* in our proof.

Contractions says that if given a *dollar* and another *dollar* you can prove a *pizza*, we may conclude that given *only one dollar* you can prove a *pizza*.

Both these rules introduce non-linearity into the logic. If we had weakening we could introduce resources into a subproof as we please, for instance if I need $40 to conclude a *lobster dinner*, but I only have a *dollar*. I could apply weakening as in the *soda* example until I had enough *quarter*s to prove the *lobster dinner*, getting money from nothing. If we had contraction we would be able to buy things cheaper than they cost, as with the *pizza* example.

Therefore we want to be careful *not* to introduce any inference rules into our logic that allow usedd to derive weakening or contraction.

3.1.2 Identity and Cut

Our first rule is an axiom. It has no conditions above the line for concluding the judgment below the line. It is the *identity* structural rule from intuitionistic logic. The *identity* states that a resource A should be sufficient to achieve A as a goal.

$$\frac{}{A \vdash A}\ \mathsf{id}_A$$

With only the resource A we can prove A. In our presentation of the rules of linear logic, the proposition to which the rule is applied is often informative in the study of the sequent calculus, hence we note this information in the subscript of the rule label.

Fundamentally the identity rule tells us if I have chocolate as a resource available to me, I can provide that to you.

Our second rule is also an axiom from intuitionistic logic. The rule is called cut, and it states the opposite of the identity rule: having achieved A as a goal, we may consume A as a resource.

$$\frac{\Delta \vdash A \quad \Delta', A \vdash C}{\Delta, \Delta' \vdash C}\ \mathsf{cut}_A$$

If using the resources Δ one can prove A and using A and Δ' as resources one can prove C, then we may conclude that under the resources Δ, Δ' we can prove C. For example cut is the idea that if I can produce pizza using my ingredients and you are hungry enough to be able to consume it to be full, we can conclude that with those ingredients you would be able to get full.

We are careful in id and cut not to allow for any additional unused resources: any resource must be used exactly once. In cut as well because the resources in the conclusion must be used exactly once either in the proof of A or C using A, we are careful to combine the resources from the subproofs, Δ and Δ'.

Note that our cut rule makes sure that we keep track of all our resources. In the conclusion, Δ and Δ' are what get consumed and C is what is produced. In the premises, Δ produces A, then A and Δ' are consumed to produce C. The net effect

is that Δ and Δ' are consumed and C is produced. This is exactly what we see in the conclusion below the line. Similarly for the id rule we had to make sure that the only resource in the context was the proposition we are proving so that all resources get used exactly once.

3.1.3 Simultaneous Conjunction

Suppose Δ can be consumed to produce A, and Δ' can be consumed to produce B. In linear logic, then, we can conclude that Δ, Δ' can be consumed to produce a package of A and B. This is written $A \otimes B$. This logic is encoded in the $\otimes$ right introduction rule which is as follows:

$$\frac{\Delta \vdash A \quad \Delta' \vdash B}{\Delta, \Delta' \vdash A \otimes B} \otimes R$$

The term $A \otimes B$ is the *simultaneous* or *multiplicative conjunction* of the two resources A and B. The resources for each subproof are combined in the conclusion but no resources may be shared between Δ and Δ' as this would constitute a violation of the ephemeral nature of propositions in linear logic. You may not have the same *pizza* in both Δ and Δ' they may both have a *pizza*, in which case the combined resources Δ, Δ' would have two *pizza*s.

We may use *simultaneous conjunction* $A \otimes B$ to package up the *premises*, i.e. the judgments above the line, of an inference into a single conclusion, making the inference a binary connective.

For an example of how we use multiple conjunction we return the spa example from Section 2.2.8. The spa wants to be able to offer a package of its *pedicure* and *massage* services. Can we prove the spa can offer this package? We know that with some resources may offer pedicures (salt baths, nail files, etc.) call these Δ and with another set of resources (oils, towels, etc.), call them Δ', they may produce a massage. Symbolically this means $\Delta \vdash \textit{pedicure}$ and $\Delta \vdash \textit{massage}$ therefore

$$\frac{\Delta \vdash \textit{massage} \quad \Delta' \vdash \textit{pedicure}}{\Delta, \Delta' \vdash \textit{massage} \otimes \textit{pedicure}} \text{ spa package}$$

Note the label of the rule $\otimes R$ denotes that this is a *right rule* which is to say that it shows how to introduce the $A \otimes B$ conjunction into the right-hand side of the sequent and achieve it as a goal. A right rule shows how to prove a proposition. Conversely, a *left rule* specifies how to use a proposition. A left rule breaks down a proposition from the set of resources on the left-hand side of the sequent. It turns out that the $\otimes$ left introduction rule tells us that if we use some context $\Delta := A_1, \ldots, A_n$ in some proof, than we just as well could have used the package $\bigotimes \Delta := A_1 \otimes A_n$, a concatenation of the resources in Δ.

$$\frac{\Delta, A, B \vdash C}{\Delta, A \otimes B \vdash C} \otimes L$$

If we may use Δ, A and B to prove C, we may use Δ and $A \otimes B$ to prove C.

3.2 Harmony

Linear logic researchers use a set of criteria for checking that the left and right rules for logical connectives satisfy both the expected intuitive and formal meanings behind them. These criteria are at the heart of linear logic and prevent us from creating left and right rules arbitrarily.

In this section we show the two important criteria for checking whether we have properly specified the left and right introduction rules for a linear logical connective. If the left and right rules meet these criteria we say that they are in *harmony*.

As a global theorem about a logic, the theorems which tell us we have a coherent logical system are cut elimination and identity. These decompose into two local properties for each connective we define, *cut reduction* and *identity expansion*. That is to say, proving that these two properties hold for every connective gives us cut elimination and identity for the whole logic.

3.2.1 Identity Expansion

We may now introduce the first criteria we apply to check that the introduction rules are consistent with each other. The idea is simple, we check that the identity property of a compound type may be reduced to other instances of the identity, without the connective we are checking. If the right and left rules match up, then we will be able to derive the simpler instances of the identity.

For example, here we check the multiplicative connective:

$$\frac{}{A \otimes B \vdash A \otimes B}\,\mathsf{id}_{A\otimes B} \quad \longrightarrow_E \quad \frac{\dfrac{\dfrac{}{A \vdash A}\,\mathsf{id}_A \quad \dfrac{}{B \vdash B}\,\mathsf{id}_B}{A, B \vdash A \otimes B}\,\otimes R}{A \otimes B \vdash A \otimes B}\,\otimes L$$

where $\rightarrow_E$ denotes the expansion, so called because we expand the proof, of an identity rule into a proof using the identity at the smaller types.

3.2.2 Cut Reduction

Similarly there is a criteria corresponding to the structural rule *cut*. Here the idea is that we may reduce a cut at some complex proposition with a connective to cuts at smaller propositions without the connective. This also checks that the right and left rules match up.

We see below, for example, a proof whose conclusion uses the cut rule for $A \otimes B$ can be replaced with a proof that instead uses the cut rule for A and the cut rule for

B.

$$\dfrac{\dfrac{\Delta \vdash A \quad \Delta' \vdash B}{\Delta, \Delta' \vdash A \otimes B}\otimes R \quad \dfrac{\Delta'', A, B \vdash C}{\Delta'', A \otimes B \vdash C}\otimes L}{\Delta, \Delta', \Delta'' \vdash C}\,\mathsf{cut}_{A\otimes B}$$

$$\longrightarrow_R$$

$$\dfrac{\Delta' \vdash B \qquad \dfrac{\Delta \vdash A \quad \Delta'', A, B \vdash C}{\Delta, \Delta'', B \vdash C}\,\mathsf{cut}_A}{\Delta, \Delta', \Delta'' \vdash C}\,\mathsf{cut}_B$$

In the above, we can interpret the proof reduction step as illustrating that there is no gain or loss of resources when we achieve $A \otimes B$ and then use it.

3.2.3 Multiplicative Unit

The unit term of binary operation $A \otimes B$ holds significance in our presentation of linear logic, so we take a moment to introduce it into our logic. The multiplicative unit is written as $\mathbf{1}$ and has the property such that $A \otimes \mathbf{1} \approx A \approx A \otimes \mathbf{1}$. We write $A \approx B$ to mean that using only A we can prove B and using only B we may prove A.

Let's inspect the right introduction rule for $\otimes$, then, and use it to inform the right introduction rule for $\mathbf{1}$. Here again is $\otimes R$ rule:

$$\dfrac{\Delta \vdash A \quad \Delta' \vdash B}{\Delta, \Delta' \vdash A \otimes B}\otimes R$$

We have two premises leading to the binary conjunction in the conclusion. So, to have a "nullary conjunction" in the conclusion, our notion of the unit term $\mathbf{1}$, we then have a inference rule with no premises as shown:

$$\dfrac{}{\cdot \vdash \mathbf{1}}\,\mathbf{1}R$$

The '$\cdot$' signifies that there are no resources, Δ is empty. Notice that we may always conclude $\mathbf{1}$.

We can use the same reasoning as above to devise the left introduction rule for $\mathbf{1}$. If we look at the rule $\otimes L$ we see that two resources in the premise get combined into the binary conjunction in the conclusion.

$$\dfrac{\Delta, A, B \vdash C}{\Delta, A \otimes B \vdash C}\otimes L \qquad \dfrac{\Delta \vdash C}{\Delta, 1 \vdash C}\,\mathbf{1}L$$

Thus, we can introduce a 1 term on the left to replace no resources listed in the premise, as follows:

$$\dfrac{\Delta \vdash C}{\Delta, 1 \vdash C}\,\mathbf{1}L$$

3.2.4 Failure of Harmony

We now have the tools necessary to demonstrate the importance of harmony. Say we had produced the following two incorrect left rules for the multiplicative conjunction

$$\frac{\Delta, A \vdash C}{\Delta, A \otimes B \vdash C} \otimes L_1?! \quad \frac{\Delta, B \vdash C}{\Delta, A \otimes B \vdash C} \otimes L_2?!$$

In attempting to show the identity expansion we see that it fails:

$$\frac{}{A \otimes B \vdash A \otimes B}\ \mathsf{id}_{A\otimes B} \quad \longrightarrow_E \quad \frac{\dfrac{\dfrac{}{A \vdash A}\ \mathsf{id}_A \qquad \dfrac{??}{\cdot \vdash B}}{A \vdash A \otimes B} \otimes R}{A \otimes B \vdash A \otimes B} \otimes L_1$$

as it would seem that B would need to be provable from nothing. In fact with these left rules we may derive

$$\frac{\dfrac{\dfrac{\Delta \vdash C}{\Delta, 1 \vdash C}\ \mathbf{1}L}{\Delta, 1 \otimes A \vdash C} \otimes L_1?!}{\Delta, A \vdash C}$$

which is to say

$$\frac{\Delta \vdash C}{\Delta, A \vdash C}\ \mathsf{weakening}$$

becomes a derived rule of inference which contradicts the basic assumptions of linear logic.

3.3 Linear implication

Traditional logic usually has a connective for implication. There we use $A \to B$ to say that "*A implies B*". One of the key aspects of traditional logic is that if we know that A is true, and we know that $A \to B$, we can conclude that B is true. This is the inference rule of modus ponens. Furthermore, A still holds as fact even after using the truth of $A \to B$.

Linear logic has a quirkier version of implication, one whose meaning carries the resource-consciousness of linear logic. In fact, linear implication is used to describe a "consumption" relation. If we may consume A to produce B, then we write $A \multimap B$. This is sometimes called the "lolli" connective because $\multimap$ bears an acute resemblance to a lollipop. Lolli has the following right introduction rule that encode its meaning:

$$\frac{\Delta, A \vdash B}{\Delta \vdash A \multimap B} \multimap R$$

If Δ and A can be used to prove B then with Δ I know that A linearly implies B. If we know that enjoying the *ambiance* of a deli and eating *sandwich* collectively provide

satisfaction, then we can also say that the enjoying *ambiance* of the deli alone sets us up so that if we consumed a *sandwich* we would have *satisfaction*. One can also see in the above that $A \multimap B$ on the right is a service contract that can be fulfilled under the assumptions in Δ. In other words, with Δ we know that "*if you give me an A you can gain a B in return*".

Let's return to the spa again, this time we will model how the spa uses you to produce money. The spa has all the resources needed to produce a massage, all the resources except for your money that is. This means that if Δ is the oils, towels, and masseuse needed for the massage then we have $\Delta, \mathit{money} \vdash \mathit{massage}$ and we may prove

$$\frac{\Delta, \mathit{money} \vdash \mathit{massage}}{\Delta \vdash \mathit{money} \multimap \mathit{massage}}\ \textsf{costly massage}$$

Now here's the lolli left introduction rule:

$$\frac{\Delta \vdash A \quad \Delta', B \vdash C}{\Delta, \Delta', A \multimap B \vdash C} \multimap L$$

This says roughly that knowing that Δ can be used to provide A and that Δ' and B can be used to provide C allows us to say that Δ, Δ', and B can be used to provide C. The left introduction rule for lolli tells us how we use this service contract, you must use some of your resources to fulfill your end of the bargain with A and you must use the rest of your resources with the B that is returned to you to achieve your goal.

Note the difference here between traditional logic and linear logic. In traditional logic we have *modus ponens* which is used to produce objectives. Given $P \to Q$ and P we may conclude Q. In our inference rule notation this would be

$$\frac{\Gamma \vdash P \quad \Gamma \vdash P \to Q}{\Gamma \vdash Q}$$

If Γ allows us to prove P and it also allows us to prove $P \to Q$ then we may also use Γ to prove Q. In linear logic we may not freely copy the assumptions of Γ if we use P to prove Q and we can prove P.

$$\frac{\Delta \vdash P \quad \Delta', P \vdash Q}{\Delta, \Delta' \vdash Q}\ \textsf{cut}$$

In the linear logic we are not free to copy the assumptions in Γ. If we use Δ to prove P we do not have access to use it for the proof of $P \multimap Q$ again. We do still have a similar rule to modus ponens in linear logic though which we see is derived from the $\multimap R$ and cut rules

$$\frac{\Delta \vdash Q \quad \dfrac{\Delta', P \vdash Q}{\Delta' \vdash P \multimap Q} \multimap R}{\Delta, \Delta' \vdash Q}$$

For another demonstration using $\multimap$ we show that our left and right rules are in harmony. First we proceed with identity expansion:

$$\frac{}{A \multimap B \vdash A \multimap B}\,\mathsf{id}_{A\multimap B} \quad \longrightarrow_E \quad \frac{\dfrac{\dfrac{}{A \vdash A}\,\mathsf{id}_A \quad \dfrac{}{B \vdash B}\,\mathsf{id}_B}{A \multimap B, A \vdash B}\multimap L}{A \multimap B \vdash A \multimap B}\multimap R$$

Now cut reduction:

$$\frac{\dfrac{\Delta, A \vdash B}{\Delta \vdash A \multimap B}\multimap R \quad \dfrac{\Delta' \vdash A \quad \Delta'', B \vdash C}{\Delta', \Delta'', A \multimap B \vdash C}\multimap L}{\Delta, \Delta', \Delta'' \vdash C}\,\mathsf{cut}_{A\multimap B}$$

$$\longrightarrow_R$$

$$\frac{\Delta'', B \vdash C \qquad \dfrac{\Delta \vdash A \quad \Delta', A \vdash B}{\Delta, \Delta' \vdash B}\,\mathsf{cut}_A}{\Delta, \Delta', \Delta'' \vdash C}\,\mathsf{cut}_B$$

Therefore we see that the left and right rules for $\multimap$ are in harmony.

3.4 Additive Conjunction

A sandwich shop can consume money to provide you with a reuben. Alternatively the sandwich shop can consume money to produce a BLT. In linear logic terms this means we can prove both

$$\text{money} \vdash \text{reuben} \quad \text{and} \quad \text{money} \vdash \text{BLT}$$

Clearly we cannot express this in linear logic with the connectives we have explained thus far as something like

$$\text{money} \vdash \text{reuben} \otimes \text{BLT}\ ?!$$

would require money to be copied in the proof. But clearly we may prove either one. The connective we use to express this situation is *additive* or *alternative conjunction* written $A \,\&\, B$, read as "*A with B*". We can achieve $A \,\&\, B$ as a goal from resources Δ exactly if we can achieve A from Δ, and we can alternatively achieve B from Δ.

$$\frac{\Delta \vdash A \quad \Delta \vdash B}{\Delta \vdash A \,\&\, B}\,\&R$$

It appears that we are copying resources in the premises, hence violating linearity, but the left rules show us that only one of the premises can be used. To consume

$A \,\&\, B$ we must choose whether we want to use A or we want to use B

$$\frac{\Delta, A \vdash C}{\Delta, A \,\&\, B \vdash C}\ \&L_1 \qquad \frac{\Delta, B \vdash C}{\Delta, A \,\&\, B \vdash C}\ \&L_2$$

The identity expansion and cut reduction show that the left and right rules are indeed in harmony and that linearity is indeed preserved. The identity expansion is

$$\frac{}{A \,\&\, B \vdash A \,\&\, B}\ \mathsf{id}_{A\&B} \quad \longrightarrow_E \quad \frac{\dfrac{\dfrac{}{A \vdash A}\ \mathsf{id}_A}{A \,\&\, B \vdash A}\ \&L_1 \quad \dfrac{\dfrac{}{B \vdash B}\ \mathsf{id}_B}{A \,\&\, B \vdash B}\ \&L_2}{A \,\&\, B \vdash A \,\&\, B}\ \&R$$

For cut reduction there are two symmetric reductions, the first one we demonstrate here and the second when the second premise of the cut is inferred with the $\&L_2$ rule.

$$\frac{\dfrac{\Delta \vdash A \quad \Delta \vdash B}{\Delta \vdash A \,\&\, B}\ \&R \quad \dfrac{\Delta', A \vdash C}{\Delta', A \,\&\, B \vdash C}\ \&L_1}{\Delta, \Delta' \vdash C}\ \mathsf{cut}_{A\&B} \quad \longrightarrow_R \quad \frac{\Delta \vdash A \quad \Delta', A \vdash C}{\Delta, \Delta' \vdash C}\ \mathsf{cut}_A$$

Using the new connective we have the sandwich rule of inference.

$$\frac{\text{money} \vdash \text{reuben} \quad \text{money} \vdash \text{BLT}}{\text{money} \vdash \text{reuben} \,\&\, \text{BLT}}\ \mathsf{sandwich}$$

3.5 Disjunction

The resource $A \,\&\, B$ provides us with a choice of either using A or B in our proof. Providing $A \,\&\, B$ we have to account for both uses, the sandwich shop must be able to make either sandwich.

If we think alternative conjunction as a resource that stands for a kind of choice, we may think of *disjunction* symmetrically. An example of disjunction is the daily special at our sandwich shop. When we walk in wanting to consume the "special" resource, we must be prepared to consume any special sandwich that might be provided. Dual to this is the sandwich shop which now gets to choose which sandwich to provide us with; either a grilled cheese or a panini.

If $A \oplus B$ is a resource we are provided with either A or B as a resource, so we have to account for either sandwich... situation.

$$\frac{\Delta, A \vdash C \quad \Delta, B \vdash C}{\Delta, A \oplus B \vdash C}\ \oplus L$$

Conversely, we may offer $A \oplus B$ by providing A or by providing B.

$$\frac{\Delta \vdash A}{\Delta \vdash A \oplus B}\ \oplus R_1 \qquad \frac{\Delta \vdash B}{\Delta \vdash A \oplus B}\ \oplus R_2$$

Concretely our sandwich shop can satisfy either

$$\text{money} \vdash \text{panini} \quad \text{or} \quad \text{money} \vdash \text{grilled cheese}$$

So we may prove

$$\frac{\text{money} \vdash \text{grilled cheese}}{\text{money} \vdash \text{grilled cheese} \oplus \text{panini}}\ \textsf{special1} \qquad \frac{\text{money} \vdash \text{panini}}{\text{money} \vdash \text{grilled cheese} \oplus \text{panini}}\ \textsf{special2}$$

the "special" sandwich rules of inference.

Chapter 4

Another Curry-Howard Isomorphism

In this section we introduce a correspondence between the process calculus we introduced in Chapter 2 and linear logic introduced in Chapter 3. This relationship between a calculus and a logic is often referred to as a Curry-Howard correspondence, named for the two researchers who independently showed a correspondence between the proof terms of natural deduction and terms of the λ-calculus. The isomorphism gives programs an operational and logical meaning, which allows us to construct programs which can easily check the soundness of a program without actually running it. The original Curry-Howard Isomorphism relates

propositions as *types*,
proofs as *programs*, and
proof normalization as *program evaluation.*

So if τ is some type or proposition from natural deduction we would label it with a term variable, such as x, and we would write $x : \tau$ meaning that the variable x has type τ.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

Combining these variables using logical connectives such as implication yields more complex expressions, called terms. You can see modus ponens in the typing rules for function application by ignoring the terms M and N and thinking of τ and σ as propositions:

$$\frac{\Gamma \vdash M : \tau \to \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash M(N) : \sigma}$$

This kind of correspondence has arisen in a number of computational settings (Wadler, 1993).

A similar connection between the π-calculus from Milner et al. (1992) and some kind of logic has been sought by programming language researchers and logicians alike.

A Curry-Howard correspondence between a process calculus and some logic is interesting for a few reasons. From the computer science perspective the correspondence may allow us to make certain guarantees about safety of our concurrent programs granted that we restrict ourselves to programming within the correspondence. For example Haskell restricts programmers to typed functional programs which decreases the programmers expressive power in their programs but these restrictions may also allow them to easily find common bugs. Programmers may use similar tools from a concurrent language correspondence that would allow them to find more complicated bugs in programs, such as network code.

The correspondence is also of interest for its own sake. To prove some complicated logical inference rule I need only construct some intuitive program. Conversely I can see what happens to my logic as I add more constructs to the programming language, observing what the new equivalences are and how they affect the coherence of my logic.

The Curry-Howard correspondence between linear logic and the π-calculus we address here is based on a line of recent research by Caires et al. (2012) with

linear propositions as *session types*,
linear proofs as *processes*, and
linear cut reduction as *communication*.

4.1 Interpreting Judgments

In the functional programming correspondence, a basic judgment of the form $M : \tau$ takes on the operational interpretation that M is a term of type τ or the logical interpretation that M is a proof of τ.

In the concurrent setting the processes are dynamic as they communicate, hence it is not meaningful to say that "*P is a process of type A*". In a stateful model of the world identity and state are decoupled from one another. For instance my identity is *Andrew*, this name is used to refer to me, but my state at the time of writing this is that of a *student*. Although my state may change to a *graduate* my identity stays the same. Similarly the identity of a process P is constant, but the services that the process offers, the *state*, changes and it is the state of the system which we reason about. The differences between *Andrew* as a *student* and *graduate* in our process setting are the services provided along channels in *Andrew*'s scope. To offer *student* services is to say that along a channel we will call *school*, *Andrew* will input *assignment*s until a *graduation* label is sent after which the *school* channel is closed, and we offer set of *graduate* services on the new *life* channel.

Processes communicate with their environment by offering services along their channels, providing an interface for other processes. So we write $P :: x : A$ to mean that the process "*P is a process that provides a service of session type A along the channel x*".

Processes not only offer services but use the services offered by other processes. To express what services a process uses we write the *sequent*

$$x_1 : A_1, \ldots, x_n : A_n \vdash P :: x : A$$

to mean that if when P is assumed to rely on communication channels offering $A_1, \ldots, A_n$ then P offers A, where $x_1, \ldots, x_n$ are all distinct. We are simply labeling the resources of linear logic sequent $\Delta \vdash A$ with channels and the goal with a process term and the channel it provides its services on.

The session type A may use any of the logical connectives we presented in the previous chapter. In the forthcoming section we shall show what each of these connectives means in terms of a process interpretation.

Of note is that our sequent formulation has a singleton right-hand side of the judgment. This notation reflects the position that offering and using services are inherently asymmetric and that we restrict processes to offering one service at a time. It also allows us to cleanly observe and enforce the scope of a name within an inference. The benefits of using this unusual version of linear logic are discussed in detail in Caires et al. (2012).

Processes evolve through interaction on channels. Once a process has interacted with a process along a channel, we may not in general use that service again along the same channel. The session types are linear propositions. We begin our discussion of the isomorphism between linear logic and our process model with a discussion of the *structural rules* for linear logic, namely how they relate to constructs of our process calculus. We then return to our process language examples to include session types, hopefully conveying their usefulness.

The correspondence allows us to either use the language of process calculi or the language of linear logic to describe process terms or linear logic process proofs because terms and proofs are essentially the same thing, up to isomorphism.

4.2 The Correspondence

4.2.1 Cut as Composition

In subsection 2.2.1 we described the *parallel composition* of two processes $(\nu x)(P \mid Q)$ as having meaning exactly when P and Q offer and use compatible services along x. Notice that the cut rule describes exactly the same thing:

$$\frac{\Delta \vdash A \quad \Delta', A \vdash C}{\Delta, \Delta' \vdash C}\ \mathsf{cut}$$

That is to say that if we can *offer* A under the resources Δ and A is *used* by some other proof to provide C, we can put the sets of resources together in Δ, Δ' and can be used to provide C. If one squints at the cut rule we see that we are simply saying that if something needs A and something else uses A, we can link them up together.

Suppose a process P offers a service of type A and suppose a process Q uses a service of type A then these two are compatible and their composition will be embodied in a new cut rule decorated with process calculus terms.

$$\frac{\Delta \vdash P :: x : A \quad \Delta', x : A \vdash Q :: z : C}{\Delta, \Delta' \vdash (\nu x)(P \mid Q) :: z : C}\ \mathsf{cut}$$

Note that the service which P offers is along channel x which is a channel that Q relies on. In linear logic, we said that A could be any of the complex propositions constructed from the proposition grammar. Similarly A here could be any complex transaction as long as P fulfills the promise of A and Q uses the promise somewhere.

4.2.2 Identity as Forwarding

Recall the id rule from linear logic. Using our new interpretation of linear propositions A, it says given the service A as a resource, we may use that resource to provide the service A. Note that this is exactly the logic governing fowarding in the process calculus, so we have the forwarding session typing rule below:

$$\frac{}{A \vdash A}\ \mathsf{id}$$

The forwarding service had access to a single service along some x and provided that same service along some other y. That is to say if we have access to a service A along x we may use that provide A along y.

$$\frac{}{x : A \vdash [x \leftrightarrow y] :: y : A}\ \mathsf{id}$$

The cut and identity of linear logic fundamentally balance the propositions on both sides of the judgment. Similarly our process interpretation of cut and identity balance offers and uses of services.

These rules cut and id *do not* describe the action of any particular services though, which we would expect as cut and identity are structural rules of the logic and are not concerned with the structure of propositions.

We will see that the harmony criteria that arise from cut and identity correspond to implementation criteria for our process expressions.

4.2.3 Input

Now we demonstrate how specific logical connectives are associated with process expression from the π-calculus. For clarity we will standardize our discussion of the logical connectives and their process counterparts by showing how to *provide* a service along a channel first. Proof-theoretically this means giving the process interpretation of the right rules of introduction first.

Recall the "lolli" connective for linear implication. To offer an $A \multimap B$, it should be true that if we are given an A then B is true:

$$\frac{\Delta, A \vdash B}{\Delta \vdash A \multimap B}\ \multimap R$$

That is to say if I offer you $A \multimap B$, you must output an A so I can return a B. Decorating the premise under the process interpretation we have

$$\Delta, y : A \vdash P :: x : B$$

which can be read as, P provides B along x, when provided A along y. We already know how to accomplish this, simply input the channel y along x,

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

Note how the inference relies on the channel x changing state. The premise says that if P uses some y offering A to provide B along x we can conclude the the input process $x(y).P$ provides $A \multimap B$ along x. Note how the state of x changes, first providing $A \multimap B$ and then providing B. The process model and linear logic are designed to capture and model this kind of state change. This is why we refer to the "type" of x as a *session type*, we are trying to encapsulate the idea that x has dynamic behavior. Once x has used an A, it may not do so again, it must behave as B. Therefore $A \multimap B$ describes a session which uses an A and behaves as B.

Using the service $A \multimap B$ is now simple to describe. If another process offers $A \multimap B$ along x, we *use* this service by making a new channel y offering A and outputting y along x.

$$\frac{\Delta \vdash P :: y : A \quad \Delta', x : B \vdash Q :: z : C}{\Delta, \Delta', x : A \multimap B \vdash (\nu y)x\langle y\rangle.(P \mid Q) :: z : C} \multimap L$$

where the channel resources Δ are used exclusively in P and the channel resources Δ' are exclusive to Q. The channel binding (νy) is necessary so that the spawned channel is not confused with any other name. It is important to take note that P and Q do not communicate with *each other* directly as they have no shared names, there is no channel linkage between them.

We demonstrate why P and Q don't necessarily communicate. Here is an example of this kind of transaction in real life.

$$(\nu\ \mathit{wallet})\ \mathit{register}\langle \mathit{wallet}\rangle.(\text{You} \mid \text{Accountant}) :: \mathit{computer} : \textsc{taxes}$$

where the process has access to some *register* : Money $\multimap$ Receipt as a resource. In English we provide money from our wallet to the register which now gives a receipt to our accountant who provides our taxes on the computer. In the process language we say that along the *register* channel we use the Money $\multimap$ Receipt service it offers. When the *register* is offering this service, You spawn a fresh *wallet* channel along the *register* upon which you will fulfill the promise of Money and your Accountant uses the Receipt now offered along *register* to do your Taxes. You and the Accountant do not communicate here, you synced up for a moment at the register but you go on about your business after.

4.2.4 Output

We have shown that *using* the service $A \multimap B$ requires output. In this section we show that that *offering* output service corresponds to multiplicative conjunction. From linear logic we have:

$$\frac{\Delta \vdash A \quad \Delta' \vdash B}{\Delta, \Delta' \vdash A \otimes B} \otimes R$$

To offer you a package of A and B together, I must be able to provide them both separately.

From the premises it is clear that the package $A \otimes B$ is offered by having one channel provide A and the other channel provide B. But how do we offer a package of services?

Recall the first spa example from 2.2.8, where we needed to provide both a MASSAGE and PEDICURE. To do so the *masseuse* introduced us to the *pedicurist* first then offered the MASSAGE service. First we had to output one of the channels along the other. This is exactly how we provide a session of type $A \otimes B$, as can be seen in the typing rule below:

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

To offer you A and B, first I may give you access along x to a y of type A which P provides and then I give you B along x. Note the similarity of the process term which uses $A \multimap B$. the process term of the left rule for linear implication. Both processes send a new channel over an existing channel to establish a new linkage between processes. We could also output a $y : B$ and have x behave as type A in the rule above. We will discuss this asymmetry below.

It should be clear how we implement a process which uses a service of type $A \otimes B$. We input a $y : A$ along x and continue using x which now offers session type B.

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

4.3 Terms as Proofs

Now that we have demonstrated the basic correspondence between linear logic inference rules and our session-typed process calculus, we examine the fruits of our labor.

In section 3.1.3 we introduced the linear logic connective $\otimes$ which showed use how to package up propositions.

$$\frac{\Delta \vdash A \quad \Delta' \vdash B}{\Delta, \Delta' \vdash A \otimes B} \otimes R \qquad \frac{\Delta, A, B \vdash C}{\Delta, A \otimes B \vdash C} \otimes L$$

Furthermore in section 4.2.3 we added process terms to these rules

$$\frac{\Delta \vdash P :: y : A \quad \Delta' \vdash Q :: x : B}{\Delta, \Delta' \vdash (\nu x) x\langle y \rangle (P \mid Q) :: x : A \otimes B} \otimes R \qquad \frac{\Delta, y : A, x : B \vdash Q :: z : C}{\Delta, x : A \otimes B \vdash x(y).Q :: z : C} \otimes L$$

A process offering $A \otimes B$ along x outputs a channel $y : A$ and then offers B along x. There is an asymmetry here as we could also have used y to provide B and x to subsequently provide A. The process which provides B and then A provides

essentially the same service as the service which provides A and then B in the sense that $A \otimes B$ can be used to provide $B \otimes A$ using a forwarding service:

$$x : A \otimes B \vdash x(y).(\nu w)z\langle w\rangle.([x \leftrightarrow w] \mid [y \leftrightarrow z]) :: z : B \otimes A$$

From a channel x offering $A \otimes B$, input the $y : A$, then we using two forwarding services composed such that the forwarding of B is first and we forward A after, swapping the order of the services. The situation here is similar to the ordering of tuples in a functional or imperative setting, the product $\tau \times \sigma$ is not equal to $\sigma \times \tau$ but the two are indeed isomorphic. For example say we have tuples of type $(\mathit{string}, \mathit{int})$ like $(\text{Jim}, 1)$ and $(\text{Jerry}, 2)$, we need only swap the order with $\mathit{swap} : (\mathit{string}, \mathit{int}) \to (\mathit{int}, \mathit{string})$

$$\mathit{swap}(\mathit{first}, \mathit{second}) = (\mathit{second}, \mathit{first})$$

to achieve an isomorphism between the two types.

Note that our Curry-Howard isomorphism now allows us to conclude the process term

$$x(y).(\nu w)z\langle w\rangle.([x \leftrightarrow w] \mid [y \leftrightarrow z])$$

corresponds to a linear logic proof of the judgment $A \otimes B \vdash B \otimes A$. We may apply the inference rules to see the proof tree:

$$\dfrac{\dfrac{\dfrac{}{x : B \vdash [x \leftrightarrow w] :: w : B}\,\mathsf{id}_B \quad \dfrac{}{y : A \vdash [y \leftrightarrow z] :: z : A}\,\mathsf{id}_A}{y : A, x : B \vdash (\nu w)z\langle w\rangle.([x \leftrightarrow w] \mid [y \leftrightarrow z]) :: z : B \otimes A}\otimes R}{x : A \otimes B \vdash x(y).(\nu w)z\langle w\rangle.([x \leftrightarrow w] \mid [y \leftrightarrow z]) :: z : B \otimes A}\otimes L$$

Stripping away the process terms we have the linear logic proof as expected:

$$\dfrac{\dfrac{\dfrac{}{B \vdash B}\,\mathsf{id}_B \quad \dfrac{}{A \vdash A}\,\mathsf{id}_A}{A, B \vdash B \otimes A}\otimes R}{A \otimes B \vdash B \otimes A}\otimes L$$

So if we wanted a proof of a fact in linear logic we could try and find the proof tree within that logic. Alternatively, we could ask a process calculus programmer to write a process term program with the appropriate session types. Using that process term and our typing rules we could unfurl that proof tree instead.

4.4 Harmony and Term Rewrites

In this section we examine the translation of the harmony criteria from linear logic to our process interpretation under our Curry-Howard correspondence. We presented the harmony criteria for linear logic in section 3.2. These criteria allowed us to check whether our constructions formed a coherent logic by balancing the resources on both sides of a sequent. The first harmony criteria was identity expansion which will give

us implementation criteria for process terms, specifically that any forwarding of any service can be implemented as the forwarding of smaller services. Cut reduction will correspond to the term reduction (τ transitions) from our process calculus. Together we see that the proof rewrites which must hold for linear logic correspond to term rewrites from the π-calculus, thus the harmony criteria which garunteed logical coherency will correspond to type preservation for our session-typed process terms.

4.4.1 Reduction

Cut reduction corresponds to process reduction and communication, the internal τ transitions discussed in our process calculus.

We demonstrate with linear implication.

$$\dfrac{\dfrac{\Delta, y : A \vdash P :: x : B}{\Delta \vdash x(y).P :: x : A \multimap B} \multimap R \quad \dfrac{\Delta' \vdash R :: w : A \quad \Delta'', x : B \vdash Q :: z : C}{\Delta', \Delta'', x : A \multimap B \vdash (\nu w)x\langle w\rangle.(R \mid Q) :: z : C} \multimap L}{\Delta, \Delta', \Delta'' \vdash (\nu x)(x(y).P \mid (\nu w)x\langle w\rangle.(Q \mid R)) :: z : C}\ \mathsf{cut}_{A \multimap B}$$

$$\longrightarrow_R$$

$$\dfrac{\Delta'' , x : B \vdash Q :: z : C \qquad \dfrac{\Delta' \vdash R :: w : A \quad \Delta, y : A \vdash P :: x : B}{\Delta, \Delta' \vdash (\nu w)(P\{w/y\} \mid R) :: x : B}\ \mathsf{cut}_A}{\Delta, \Delta', \Delta'' \vdash (\nu x)((\nu w)(P\{w/y\} \mid R) \mid Q) :: z : C}\ \mathsf{cut}_B$$

Note the conclusion before and after the reduction are

$$\begin{array}{l} \Delta, \Delta', \Delta'' \vdash (\nu x)(x(y).P \mid (\nu w)x\langle w\rangle.(Q \mid R)) :: z : C \\ \xrightarrow{\quad\tau\quad} \\ \Delta, \Delta', \Delta'' \vdash (\nu x)((\nu w)(P\{w/y\} \mid R) \mid Q) :: z : C \end{array}$$

We apply the structural congruences of the π-calculus, extruding the bindings on x and w and invoking associativity, to make these easier to read:

$$\begin{array}{l} \Delta, \Delta', \Delta'' \vdash (\nu x)(\nu w)(x(y).P \mid x\langle w\rangle.(R \mid Q)) :: z : C \\ \xrightarrow{\quad\tau\quad} \\ \Delta, \Delta', \Delta'' \vdash (\nu x)(\nu w)(P\{w/y\} \mid R \mid Q) :: z : C \end{array}$$

Indeed we see that cut reduction mirrors a process reduction, matching an input with a corresponding output, modulo the structural rules. The general case of this reduction is just the τ transition for input/output:

$$(x(y).P \mid x\langle w\rangle.Q) \xrightarrow{\quad\tau\quad} P\{w/y\}\ Q$$

4.4.2 Expansion

Cut reduction shows how a server and client of a service communicate with each other. Identity expansion on the other hand tells us that forwarding a complex service should be decomposable into forwardings of channels of the simpler types.

$$\frac{}{x : A \multimap B \vdash [x \leftrightarrow z] :: z : A \multimap B}\ \mathsf{id}_{A \multimap B}$$

$$\longrightarrow_E$$

$$\frac{\dfrac{\dfrac{}{y : A \vdash [y \leftrightarrow w] :: w : A}\ \mathsf{id}_A \quad \dfrac{}{x : B \vdash [x \leftrightarrow z] :: z : B}\ \mathsf{id}_B}{x : A \multimap B, y : A \vdash (\nu w)x\langle w\rangle.([y \leftrightarrow w] \mid [x \leftrightarrow z]) :: z : B} \multimap L}{A \multimap B \vdash A \multimap Bx : A \multimap B \vdash z(y).(\nu w)x\langle w\rangle.([y \leftrightarrow w] \mid [x \leftrightarrow z]) :: z : A \multimap B} \multimap R$$

In words we have $z : A \multimap B$ mimic $x : A \multimap B$ directly, or we could equivalently input a $y : A$ along z and output a new w to x, where w mimics $y : A$ and z now mimics $x : B$.

4.5 Some Fine Print

4.5.1 Composing Cut and Identity

Composing a forwarding service $[x \leftrightarrow y]$ of A and a process Q which uses $y : A$ should be equivalent to just renaming the channel which Q uses to provide A, $Q\{x/y\}$ provide the name x doesn't occur in Q. In the correspondence this means that a cut with the identity should have no effect. We show this using the process terms in the following inference

$$\frac{\dfrac{}{y : A \vdash [y \leftrightarrow x] :: x : A}\ \mathsf{id} \quad \Delta, x : A \vdash Q :: z : C}{\Delta, y : A \vdash (\nu x)([y \leftrightarrow x] \mid Q) :: z : C}\ \mathsf{cut}$$

$$\longrightarrow$$

$$\Delta, y : A \vdash Q\{y/x\} :: z : C$$

And symmetrically,

$$\frac{\Delta \vdash P :: x : C \quad \dfrac{}{x : C \vdash [x \leftrightarrow z] :: z : C}\ \mathsf{id}}{\Delta \vdash (\nu x)(P \mid [x \leftrightarrow z]) :: z : C}\ \mathsf{cut}$$

$$\longrightarrow$$

$$\Delta \vdash P\{x/z\} :: z : C$$

Thus composition with a forwarding service, the *identity*, is somehow equivalent to renaming a channel, "cutting out the middleman". The reduction arrows here are not the internal transition arrows from our process calculus, $\overset{\tau}{\not\to}$, but more like *structural reductions*, which is to say that no communication or computation is taking place. A *structural reduction* between two process terms can be viewed as a difference in implementation. For example, say you have a *work* and *personal* email address. On the personal email service you offer the *respond* service, but everyone at work only knows your *work* email so they send emails there and you have Google forward those emails to *personal*. We can say that you are essentially the same as if you offered to *respond* over your *work* email, modulo a *structural reduction*.

4.5.2 Composing Cuts

If we have several process terms composed together, it would be desirable if they needed only to synchronize when their interactions required not on every computation. If Heather, Sammy, and Max a having independent conversations, the order in which they call each other should not matter.

In our correspondence, this is reflected in the fact that the order of consecutive cuts are insignificant. This is established by using the fundamental *structural equivalences* of process terms:

$$\begin{array}{rcll} (P \mid Q) \mid R & \equiv & P \mid (Q \mid R) & \text{associativity} \\ P \mid Q & \equiv & Q \mid P & \text{commutativity} \\ P \mid (\nu x)Q & \equiv & (\nu x)(P \mid Q) & \text{scope extrusion} \\ & & & \text{where } x \in \mathsf{fn}(P) \end{array}$$

Where the condition $x \notin \mathsf{fn}(P)$ means that variable x is not a free name in P. Since (νx) binds a name, this condition ensures that two different variables with the same name are not confused. If we find a conflict we may rename the bound variable to allow for scope extrusion.

These basic equivalences may then be used to derive the corresponding laws which we will consider to be higher order *structural congruences* or *structural equivalences*.

$$\begin{array}{rcl} (\nu x)((\nu y)(P \mid Q) \mid R) & \equiv & (\nu y)(P \mid (\nu x)(Q \mid R)) \\ & & \text{where } x \notin \mathsf{fn}(P) \text{ and } y \notin \mathsf{fn}(R) \\ (\nu x)(P \mid (\nu y)(Q \mid R)) & \equiv & (\nu y)(Q \mid (\nu x)(P \mid R)) \\ & & \text{where } x \notin \mathsf{fn}(Q) \text{ and } y \notin \mathsf{fn}(P) \end{array}$$

For our purposes here, we distinguish processes modulo structural congruence. A practical disadvantage of this construction comes when implementing an algorithm for type-checking processes, because we may need to rearrange expressions by structural congruences, before applying the typing rule makes sense.

4.6 Extending the Correspondence

We will now extend the session-typed processes to include termination corresponding to the multiplicative unit **1** and choice which will correspond to logical connectives

$\oplus$ and $\&$.

4.6.1 Termination

We claimed in section 2.2.4 that it would be beneficial to discuss the multiplicative unit $\mathbf{1}$. Here the discussion pays off as we see that $\mathbf{1}$ plays a role as the termination service.

Just as we derived the right and left rules for $\mathbf{1}$ thinking of it as the nullary version of $\otimes$ we may derive the process interpretation rules by thinking of $\mathbf{1}$ as the process which then outputs nothing and has no continuation:

$$\frac{}{\cdot \vdash x\langle\rangle.\mathbf{0} :: x : 1}\ \mathbf{1}R$$

We discussed this process in section 2.2.4 in part due to this correspondence with linear logic. When using a termination service we expect that we will input nothing, signaling the connection is terminated.

$$\frac{\Delta \vdash Q :: z : C}{\Delta, x : \mathbf{1} \vdash x().Q :: z : C}\ \mathbf{1}L$$

Cut reduction here corresponds to the process reduction

$$(\nu x)(x\langle\rangle.\mathbf{0} \mid x().Q) \xrightarrow{\tau} (\mathbf{0} \mid Q) \equiv Q$$

The left and right rules defined for $\mathbf{1}$ are sufficient towards the goal of establishing a Curry-Howard isomorphism between linear logic and our process calculus. We may not need a full Curry-Howard isomorphism to be able to make the kinds of guarantees we want for a practical language though.

The idea here is that a process P which does not offer any services is typed as $\Delta \vdash P :: x : 1$. Once composed processes offering the appropriate resources required by Δ, it should evolve by internal actions only and be closed to communication. Composing P with another closed process $\Delta' \vdash Q :: z : 1$ should have no effect and both should evolve independently, without interacting. Currently our rules do not allow for the parallel composition of P and Q, $P \mid Q$, observe

$$\frac{\Delta \vdash P :: x : \mathbf{1} \qquad \dfrac{\Delta' \vdash Q :: z : \mathbf{1}}{\Delta' x : \mathbf{1} \vdash x().Q :: z : \mathbf{1}}\ \mathbf{1}L}{\Delta, \Delta' \vdash (\nu x)(P \mid x().Q) :: z : \mathbf{1}}\ \mathsf{cut}$$

The restriction $x().Q$ on Q forces the composition to be *sequential* and not *parallel* in that P must finish before Q can continue.

We might be willing to give up some of the correspondence to recover parallelism from the model. There are several ways that we could go about this and here we pursue this goal by making the $\mathbf{1}L$ rule silent such that we have the same proof term in the premise and conclusion Pfenning (2012). With this alternate rule different

proofs collapse to the same process, so we no longer have an isomorphism. This is a phenomenon which occurs in other applications of the Curry-Howard Isomorphism and we say that proofs are *contracted* to programs. Consider the alternate left and right rules for $\mathbf{1}$ as follows

$$\frac{}{\cdot \vdash \mathbf{0} :: x : 1}\ (\mathbf{1}R)' \qquad \frac{\Delta \vdash Q :: z : C}{\Delta, x : \mathbf{1} \vdash Q :: z : C}\ (\mathbf{1}L)'$$

The cut reduction is then just the structural congruence $(\nu x)(\mathbf{0} \mid Q) \equiv Q$ where $x \notin \mathsf{fn}(Q)$. Note that now we have

$$\frac{\Delta \vdash P :: x : \mathbf{1} \quad \dfrac{\Delta' \vdash Q :: z : \mathbf{1}}{\Delta' x : \mathbf{1} \vdash Q :: z : \mathbf{1}}\ \mathbf{1}L}{\Delta, \Delta' \vdash (\nu x)(P \mid Q) :: z : \mathbf{1}}\ \mathsf{cut}$$

Provided we have $x \notin \mathsf{fn}P$ we can use the structural congruence $(\nu x)P \equiv P$ to turn the process in the conclusion into $(P \mid Q)$, proving an *parallel* composition rule.

From a practical standpoint we may want to allow for both kinds of termination in our programming language. The implementation of "channels" in *GoLang*, allows for synchronous or asynchronous termination but only the server may signal termination along the channel (Pike & Gerrand, 2013).

If we are not programming in a setting with a concurrent type theory, such as an asynchronous or strongly parallel setting, there are different control structures, for example the *GoLang* select statement (Pike & Gerrand, 2013), benefit from the CLOSE/WAIT syntax, while for other design patterns, such as an HTML server, it might be costly to have to wait for a close signal before moving on with computation.

4.6.2 Choice

External choice or branching means to offer a service which provides a choice between the services A and B. The Cathy process in 2.1 offers external choice because the user must decide whether to send an email, a phone call or drop-in visit. The traditional presentation of *external choice* in the literature Caires et al. (2012) uses *binary guarded choice* in the process calculus to establish the correspondence between the linear connective $A \,\&\, B$ with its process interpretation. This is to say that instead of the *indexed labeled choice* process we defined in Chapter 2 $x.\text{CASE}(\overline{l_i \Rightarrow P_i})$ we use the *binary guarded choice* is written as $x.\text{CASE}(P, Q)$ and the selection processes $x.\mathsf{inl}$, "in left" to choose the left process P, and $x.\mathsf{inr}$, "in right" to choose the right-hand process Q. It is very easy to implement the indexed version using the binary guarded choice but it is easier to demonstrate the process interpretation of the left and right rules for $A \,\&\, B$ that we have already defined.

Recall the left and right rules for the external choice connective:

$$\frac{\Delta \vdash A \quad \Delta \vdash B}{\Delta \vdash A \,\&\, B}\ \&R$$

$$\frac{\Delta, A \vdash C}{\Delta, A \,\&\, B \vdash C}\,\&L_1 \qquad \frac{\Delta, B \vdash C}{\Delta, A \,\&\, B \vdash C}\,\&L_2$$

At the client's discretion only either A or B will be used, and therefore the context Δ can safely be propagated to both premises. Conversely, if a client wants use $A \,\&\, B$ from the environment, the must choose either A or B.

A client indicates which service they chose by sending a label (inl or inr) along a channel and then the client uses uses the service they selected A or B respectively along that same channel.

$$\frac{\Delta \vdash P :: x : A \quad \Delta \vdash Q :: x : B}{\Delta \vdash x.\textsc{case}(P, Q) :: x : A \,\&\, B}\,\&R$$

$$\frac{\Delta, x : A \vdash Q :: z : C}{\Delta, x : A \,\&\, B \vdash x.\mathsf{inl}; Q :: z : C}\,\&L_1 \qquad \frac{\Delta, x : B \vdash Q :: z : C}{\Delta, x : A \,\&\, B \vdash x.\mathsf{inr}; Q :: z : C}\,\&L_2$$

A curiosity here is the difference in notation between i/o and choice. With i/o we notate the continuation of a process with "." for example $x\langle y\rangle.Q$ means to "input a y along x and continue as Q". On the other hand with choice we use the "." to mean the sending of a label along that channel, $x.\mathsf{inl}; Q$ means to send the inl label along x and continue as Q. It is left to the reader to decide whether the distinction in the syntax for sending a label opposed to other input is useful, or if overloading the syntax $x\langle\mathsf{inl}\rangle$ would be clearer.

Cut reduction corresponds to the following process reduction

$$(\nu x)(x.\mathsf{inl}; R \mid x.\textsc{case}(P, Q)) \longrightarrow^{\tau} (\nu x)(R \mid P)$$
$$(\nu x)(x.\mathsf{inr}; R \mid x.\textsc{case}(P, Q)) \longrightarrow^{\tau} (\nu x)(R \mid Q)$$

Identity expansion for $x : A \,\&\, B \vdash [x \leftrightarrow z] :: z : A \,\&\, B$ corresponds to the implementation

$$[x \leftrightarrow z] \Rightarrow z.\textsc{case}(x.\mathsf{inl}; [x \leftrightarrow z] \mid x.\mathsf{inr}; [x \leftrightarrow z])$$

Internal choice, often just called choice in the literature, means the server offers either A or B along a channel x. The distinction between branching and choice is that in branching the user process decides but in choice the offering process is the one to decide. Choice is symmetric to branching in the sense that now the user must be prepared for either A or B. The process interpretation of the rules for $A \oplus B$ should be clear from the *external choice* choice section above.

$$\frac{\Delta \vdash P :: x : A}{\Delta \vdash x.\mathsf{inl}; P :: x : A \oplus B}\,\oplus R_1 \qquad \frac{\Delta \vdash P :: x : B}{\Delta \vdash x.\mathsf{inr}; P :: x : A \oplus B}\,\oplus R_2$$

$$\frac{\Delta, x : A \vdash P :: z : C \quad \Delta, x : B \vdash Q :: z : C}{\Delta, x : A \oplus B \vdash x.\textsc{case}(P, Q) :: z : C}\,\oplus L$$

The reduction here is the same as the reduction for *external choice* and the identity expansion for $x : A \oplus B \vdash [x \leftrightarrow z] :: z : A \oplus B$ corresponds to the implementation

$$[x \leftrightarrow z] \Rightarrow x.\textsc{case}(z.\mathsf{inl}; [x \leftrightarrow z] \mid z.\mathsf{inr}; [x \leftrightarrow z])$$

4.7 Taking Stock

The grammar for the session types and corresponding processes we have defined gives us the following grammar:

Session Types	A, B, C	$::=$	$A \multimap B$	input channel of type A and continue as B
		$\mid$	$A \otimes B$	output a fresh channel of type A and continue as B
		$\mid$	$\mathbf{1}$	terminate
		$\mid$	$\& \{\overline{l_j : A_j}\}$	offer a choice between l_j and continue as A_j
		$\mid$	$\oplus \{\overline{l_j : A_j}\}$	provide one of the l_j and continue as A_j
Processes	P, Q	$::=$	$x\langle M\rangle.P$	term output
		$\mid$	$x\,(y)\,.P$	input
		$\mid$	$(\nu y)x\langle y\rangle.P$	bound output
		$\mid$	$(\nu y)\,.P$	name restriction
		$\mid$	$P \mid Q$	parallel composition
		$\mid$	$x.\textsc{case}\,(\overline{l_j \Rightarrow P_j})$	branching
		$\mid$	$x.l_i; P$	selection
		$\mid$	$[x \leftrightarrow y]$	forwarding
		$\mid$	$\mathbf{0}$	termination

Chapter 5

Quantification

Note that in the beginning of Chapter 3 we parametrized an atomic proposition *student* with the name x, meaning for all students x, where x is a *string* such as *"Jim"* or *"Jerry"*. This an example of an extension of linear logic in which we quantify over all x which are a proof of a *string* type.

A typed process calculus serves much more practical significance if we are able to integrate data from a typed functional or imperative language. Allowing processes to pass integers, strings, lists, and other basic data types is invaluable to a useful process model, we do not want to worry about constructing the Church numeral equivalents in channels. To this end we introduce quantification into linear logic, where the domains of the quantifiers are from the typed λ-calculus or some other well typed calculus, although we will show there are very few restrictions needed of these types and terms for them to make sense as an extension, i.e. harmonize in our linear logic.

This requires that we have a new external typing judgment in the environments of our inferences.

A natural question that arises in our considerations of linear logic, is how we can extend our logic to reason about linear propositions which are parametrized by proofs from some nonlinear logic. We present an extension of linear logic with quantification, where the domains of the quantifiers are external to linear logic.

This requires us to have a new external typing judgment.

5.0.1 Judgments

In order to introduce these quantifiers we must add a new external typing judgment to our inference rules. The basic judgment is of the form $M{:}\tau$, meaning the object M is a proof of type τ. We will interchangeably refer to objects as *terms* (viewing these objects as functional expressions) and as *values* (contrasting them with channels). We will use this basic judgment in its hypothetical form:

$$\Psi \vdash M : \tau$$

where Ψ is a collection of distinct hypotheses $n_i{:}\tau_i$, analogous to Δ from the linear logic judgments. The context Ψ is where we record the types for the term variables allowing us to be explicit about the available parameters for the proof. These term

variables are *not* considered resources and may be used arbitrarily often. Few requirements are needed of the types τ and we will present the necessary restrictions for the quantifiers to make sense, i.e. for the left and right rules to harmonize. Otherwise our presentation shows that it does not really matter how the types and the terms which prove them are constructed.

The general form of our judgment becomes:

$$\underbrace{m_1{:}\tau_1, \ldots, m_k{:}\tau_k}_{\Psi} \; ; \; \underbrace{x_1{:}A_1, \ldots, x_i{:}A_i}_{\Delta} \vdash P :: z : C$$

where all variables are distinct, m are term variables and x are linear channels.

5.1 Universal Quantification

We first examine universal quantification. We write $\forall n{:}\tau.A$, universally quantifying over objects which are a proof of type τ.

Recall that the left introduction rule for $\forall n{:}\tau.A$ tells us how to use this resource. The proposition $\forall n{:}\tau.A$ means that A is true for any object of type τ. So when we can use our term variables to prove some $M{:}\tau$, we may substitute all occurrences of n with M in the A which uses it:

$$\frac{\Psi \vdash M : \tau \quad \Psi \; ; \; \Delta, A\{M/n\} \vdash C}{\Psi \; ; \; \Delta, \forall n{:}\tau.A \vdash C} \; \forall L$$

Here, the typing of M does not depend on Δ, because we stipulate that terms cannot depend on linear resources, i.e. functional variables cannot rely on the channels from processes.

The right rule shows how we prove $\forall n{:}\tau.A$ is true. If we can prove $A\{m/n\}$ for any new parameter m of type τ then we may provide $\forall n{:}\tau.A$:

$$\frac{\Psi, m{:}\tau \; ; \; \Delta \vdash A\{m/n\}}{\Psi \; ; \; \Delta \vdash \forall n{:}\tau.A} \; \forall R$$

We may only apply the right rule if m is not already in Ψ, but we are always able to choose a fresh name if a copy is needed. Note that for n to appear in some A we must allow some atomic propositions to depend on term variables, such as the *student*(x) proposition.

5.2 Harmony for Universal Quantification

We will now show what logical restrictions on the term variables and types, or rather the judgment $\Psi \vdash M{:}\tau$, are needed to make sure our left and right rules for universal quantification are consistent. Consider the cut reduction criterion first:

$$\frac{\dfrac{\Psi, m{:}\tau \; ; \; \Delta \vdash A\{m/n\}}{\Psi \; ; \; \Delta \vdash \forall n{:}\tau.A} \; \forall R \quad \dfrac{\Psi \vdash M : \tau \quad \Psi \; ; \; \Delta', A\{M/n\} \vdash C}{\Psi \; ; \; \Delta', \forall n{:}\tau.A \vdash C} \; \forall L}{\Psi \; ; \; \Delta, \Delta' \vdash C} \; \mathsf{cut}_{\forall n:\tau.A}$$

We need to be able to reduce this complex cut to the cut at the two premises: $A\{m/n\}$ in the first premise, and $A\{M/n\}$ in the second premise. We must *substitute* M for m throughout the sequent $\Psi, m{:}\tau \ ; \ \Delta \vdash A\{m/n\}$. Since m is a fresh name, thus does not occur in Γ, Δ or A, we obtain $\Psi \ ; \ \Delta \vdash A\{M/n\}$. Therefore, for the first restriction on the types and terms we have:

> **Substitution Principle.** If $\Psi \vdash M : \tau$ and $\Psi, m{:}\tau \ ; \ \Delta \vdash A$ then $\Psi \ ; \Delta \vdash A\{M/m\}$, provided Δ does not depend on m.

This principle gives us the following *admissible rule*, rather than a rule of inference,

$$\frac{\Psi \vdash M : \tau \quad \Psi, m{:}\tau \ ; \ \Delta \vdash A}{\Psi \ ; \ \Gamma\{M/m\} \ ; \ \Delta\{M/m\} \vdash A\{M/m\}} \ \mathsf{subst}$$

Where whenever we can prove the premises, then we can construct the conclusion based on our assumptions about how $\Psi \vdash M : \tau$ interacts with sequents. The dashed line indicates we are using some principle about proofs in general rather than some derived rule from the basic axioms. Using this notation, the evidence for cut reduction is:

$$\frac{\dfrac{\Psi, m{:}\tau \ ; \ \Delta \vdash A\{m/n\}}{\Psi \ ; \ \Delta \vdash \forall n{:}\tau.A} \ \forall R \qquad \dfrac{\Psi \vdash M : \tau \quad \Psi \ ; \ \Delta', A\{M/n\} \vdash C}{\Psi \ ; \ \Delta', \forall n{:}\tau.A \vdash C} \ \forall L}{\Psi \ ; \ \Delta, \Delta' \vdash C} \ \mathsf{cut}_{\forall n:\tau.A}$$

$$\longrightarrow_R$$

$$\frac{\dfrac{\Psi \vdash M : \tau \quad \Psi, m{:}\tau \ ; \ \Delta \vdash A}{\Psi \ ; \ \Delta \vdash A\{M/n\}} \ \mathsf{subst} \qquad \Psi \ ; \ \Delta', A\{M/n\} \vdash C}{\Psi \ ; \ \Delta, \Delta' \vdash C} \ \mathsf{cut}_{A\{M/n\}}$$

Here $\Delta\{M/n\} = \Delta$ by the freshness of m in the $\forall R$ rule.

For the identity expansion criterion, recall that we attempt to reduce an application of the identity rule at a complex type to the identity rule at the smaller types. We have:

$$\frac{}{\cdot \ ; \ \forall n{:}\tau.A \vdash \forall n{:}\tau.A} \ \mathsf{id}_{\forall n:\tau.A} \longrightarrow_E \frac{\dfrac{\dfrac{}{m{:}\tau \vdash m : \tau} \ \mathsf{hyp} \qquad \dfrac{}{m{:}\tau \ ; \ A\{m/n\} \vdash A\{m/n\}} \ \mathsf{id}_{A\{m/n\}}}{m{:}\tau \ ; \ \forall n{:}\tau.A \vdash A\{m/n\}} \ \forall L}{\cdot \ ; \ \forall n{:}\tau.A \vdash \forall n{:}\tau.A} \ \forall R$$

We see that for identity expansion to make sense we need some sort of identity for term variables and types. We assume a hypothesis principle for the external typing judgments of terms. Usually, this is simply an inference rule, but because this typing judgment is external to linear logic, we do not want to fix it as a rule.

> **Hypothesis.** $m{:}\tau \vdash m : \tau$ for any variable m and type τ

In order to generalize the identity rule from the sequent calculus, for the arbitrary hypotheses Ψ to appear in the rule, we must be able to *weaken* any judgment with new typing assumptions $m{:}\tau$, because they are not necessarily used. We apply this principle as necessary without mention.

Weakening. If $\Psi \,;\, \Delta \vdash C$ then $\Psi, m{:}\tau \,;\, \Delta \vdash C$.

Here we suppose that m is fresh, not already declared in Ψ, so that no variable is declared more than once in a judgment. The term typing judgment itself must also have internal substitution and weakening principles for this discussion to be consistent.

5.3 Existential Quantification

Existential quantification $\exists n{:}\tau.A$ can be considered dual to $\forall n{:}\tau.A$. In order to prove $\exists n{:}\tau.A$, we have to supply some term M of the correct type.

$$\frac{\Psi \vdash M : \tau \quad \Psi \,;\, \Delta \vdash A\{M/n\}}{\Psi \,;\, \Delta \vdash \exists n{:}\tau.A}\ \exists R$$

In order to use it, we have to suppose we have some new parameter $m{:}\tau$.

$$\frac{\Psi, m{:}\tau \,;\, \Delta, A\{m/n\} \vdash C}{\Psi \,;\, \Delta, \exists n{:}\tau.A \vdash C}\ \exists L$$

As usual we choose m fresh so that it does not appear in Δ, A, and C.

5.3.1 Example: Sandwich(x)

For an example of how we parametrize how we use quantifiers, let's revisit the sandwich rule of inference from 3.4:

$$\frac{\text{money} \ \vdash\ \text{reuben} \quad \text{money} \ \vdash\ \text{BLT}}{\text{money} \vdash \text{reuben} \ \&\ \text{BLT}}\ \textsf{sandwich}$$

Let's complicate the transaction a little. Let Ψ be a list of our preferences:

$$\Psi := \text{sandwich preference} : \textit{sandwich style}, \text{pizza preference} : \textit{pizza style}, \ldots$$

then with money we may have this sandwich:

$$\frac{\Psi \vdash \text{sandwich preference} : \textit{sandwich style} \quad \Psi \,;\, \text{money} \vdash \text{sandwich}(\text{sandwich preference})}{\Psi \,;\, \text{money} \vdash \exists x{:}\textit{sandwich style}.\text{sandwich}(x)}$$

5.4 Term Passing

We will show that with the Curry-Howard interpretation the quantified linear propositions will be the passing of basic data (terms), rather than channels. The universal quantifier, the "for all" quantifier, corresponds to data input, while the existential quantifier represents data output. The result is somewhat reminiscent of the applied π-calculus Abadi & Fournet (2001).

All the sequent judgments have been generalized by adding new hypotheses within the context Ψ, assigning types to term variables, written as

$$\Psi \; ; \; \Delta \vdash P :: x : A$$

The hypotheses Ψ are propagated to all premises in all inference rules we have presented thus far. We may freely do this by the assumptions we placed on the types and inhabiting terms of the judgment $\Psi \vdash M : \tau$. For example, the identity and cut rules are now

$$\frac{}{\Psi \; ; \; x{:}A \vdash [x \leftrightarrow z] :: z : A} \; \mathsf{id}_A$$

$$\frac{\Psi \; ; \; \Delta \vdash P :: x : A \quad \Psi \; ; \; \Delta', x{:}A \vdash Q :: z : C}{\Psi \; ; \; \Delta, \Delta' \vdash (\nu x)(P \mid Q) :: z : C} \; \mathsf{cut}_A$$

5.4.1 Term Input

We model the input of terms with universal quantification. To provide the $\forall n{:}\tau.A$ service means to provide A once we get an $m{:}\tau$. For the right rule the process term offers to input a term and then behave as A with the term we input, say some m, substituted throughout A.

$$\frac{\Psi, m{:}\tau \; ; \; \Delta \vdash P\{m/n\} :: x : A\{m/n\}}{\Psi \; ; \; \Delta \vdash x(n).P :: x : \forall n{:}\tau.A} \; \forall R$$

Just as with channel i/o, the channel on which we interact with the process changes the service which it offers. Also the $\forall L$ rule must therefore correspond to the process expression which provides the output to match the $\forall R$ rule, in order for the rules to harmonize:

$$\frac{\Psi \vdash M : \tau \quad \Psi \; ; \; \Delta', x{:}A\{M/n\} \vdash Q :: z : C}{\Psi \; ; \; \Delta', x{:}\forall n{:}\tau.A \vdash \overline{x}\langle M \rangle.Q :: z : C} \; \forall L$$

Just as in our discussion of quantifiers in the previous section, m must be chosen fresh so that the context $\Psi, m{:}\tau$ makes sense in the $\forall R$ rule, where we may always rename to avoid conflict. We overload our substitution notation to allow for substitution of names by terms, writing $\{M/y\}$ for the capture-avoiding substitution of M for y in A.

Reduction

Applying cut to the conclusions of right and left rules above we have:

$$\frac{\Psi\ ;\ \Delta \vdash x(n).P :: x : \forall n{:}\tau.A \Psi\ ;\ \Delta', x{:}\forall n{:}\tau.A \vdash \overline{x}\langle M\rangle.Q :: z : C}{\Psi\ ;\ \Delta, \Delta' \vdash (\nu x)(x(n).P \mid \overline{x}\langle M\rangle.Q) :: z : C}\ \mathsf{cut}_\forall$$

For the quantifiers to be logically consistent they must satisfy the substitution principle so that we may substitute M for n in the premise of the $\forall R$, thereby justifying the cut reduction. After that we obtain the following cut at the smaller types:

$$\frac{\begin{array}{l}\Psi\ ;\ \Delta \vdash P\{M/n\} :: x : A\{M/n\} \\ \Psi\ ;\ \Delta', x{:}A\{M/n\} \vdash Q :: z : C\end{array}}{\Psi\ ;\ \Delta, \Delta' \vdash (\nu x)(P\{M/n\} \mid Q) :: z : C}\ \mathsf{cut}$$

from which we see the usual i/o process reduction

$$(\nu x)(x(n).P \mid \overline{x}\langle M\rangle.Q) \longrightarrow (\nu x)(P\{M/n\} \mid Q)$$

In other words, we may pass terms of some functional language in the π-calculus, in addition to passing names of channels as usual.

5.4.2 Term Output

A channel $x : \exists y{:}\tau.A$ should offer the service symmetric to the term input service. Term output along x means to offer to output a term M of type τ along x and then offer $A\{M/y\}$. Again, this is dual to term input with type $\forall y{:}\tau.A$.

$$\frac{\Psi \vdash M : \tau \quad \Psi; \Delta \vdash P :: x : A\{M/n\}}{\Psi; \Delta \vdash \overline{x}\langle M\rangle.P :: x : \exists n{:}\tau.A}\ \exists R$$

$$\frac{\Psi, m{:}\tau; \Delta', x{:}A\{m/n\} \vdash Q\{m/n\} :: z : C}{\Psi; \Delta', x{:}\exists n{:}\tau.A \vdash x(n).Q :: z : C}\ \exists L$$

Applying cut to the conclusions of these two rules yields:

$$\frac{\Psi; \Delta \vdash \overline{x}\langle M\rangle.P :: x : \exists n{:}\tau.A \Psi; \Delta', x{:}\exists n{:}\tau.A \vdash x(n).Q :: z : C}{\Psi; \Delta, \Delta' \vdash (\nu x)(x\langle M\rangle.P \mid x\langle n\rangle.Q) :: z : C}\ \mathsf{cut}$$

which, appealing to the substitution property as before in the premise of $\exists L$, reduces to the same communication via terms as with the universal quantifier pair (modulo structural congruences in the π-calculus).

$$(\nu x)(\overline{x}\langle M\rangle.P \mid x\langle n\rangle.Q) \longrightarrow (\nu x)(P \mid Q\{M/n\})$$

5.5 Taking Stock

We now add quantification to our session types and term input/output to our processes:

Session Types	A, B, C	$::=$	$A \multimap B$	input channel of type A and continue as B
		$\mid$	$A \otimes B$	output a fresh channel of type A and continue as B
		$\mid$	$\mathbf{1}$	terminate
		$\mid$	$\& \{\overline{l_j : A_j}\}$	offer a choice between l_j and continue as A_j
		$\mid$	$\oplus \{\overline{l_j : A_j}\}$	provide one of the l_j and continue as A_j
		$\mid$	$\tau \supset A$	input a term of type τ and continue as A
		$\mid$	$\tau \wedge A$	output a term of type τ and continue as A
Terms	M, N	$::=$	$M\ N \mid (M, N) \mid \ldots$	(basic data constructors)
Processes	P, Q	$::=$	$x\langle M\rangle.P$	term output
		$\mid$	$x\,(y)\,.P$	input
		$\mid$	$(\nu y)x\langle y\rangle.P$	bound output
		$\mid$	$(\nu y)\,.P$	name restriction
		$\mid$	$P \mid Q$	parallel composition
		$\mid$	$x.\textsc{case}\,(\overline{l_j \Rightarrow P_j})$	branching
		$\mid$	$x.l_i; P$	selection
		$\mid$	$[x \leftrightarrow y]$	forwarding
		$\mid$	$\mathbf{0}$	termination

Conclusion

We have explored how session-typed process terms correspond to the inference rules for connectives in linear logic.

In Chapter 2 we gave the specification for a synchronous π-calculus which gave us a means to naturally describe concurrent programs. In Chapter 3 we introduced linear logic, the logic of resources, in which truth is consumed to make inferences. We then demonstrated that the session-typed calculus we presented was in one-to-one correspondence with linear logic. For example cut reduction, the computational workhorse of linear logic, corresponded to communication in the form of process term reductions. The harmony critera for our logic corresponded to type preservation for our process terms.

This correspondence is interesting in its own right. That a process calculus is isomorphic to some kind of logic means that process terms are simply proofs of some logical proposition. The isomorphism also means that global theorems about linear logic hold for our process terms, in fact Toninho et al. (2014) use the correspondence to guarantee that the process calculus is free from deadlock by using nondivergence theorems from linear logic.

The proliferation of system components such as objects-stores, GPS services, relational databases, load balancers, and the plentiful variety of other services has made concurrency support almost required for a programming language. Modern databases such as *Riak*, programmed in Erlang, or *Doozer* and *etcd*, programmed in GoLang, take advantage of concurrency primitives. The programmers who make these systems take advantage of distributed systems research to make consensus guarantees about their database protocols but the concurrency issues in the system code can still be difficult to find and pin down. If we could augment a concurrent programming language system to include formal specifcation and verification components in the form of a session type inference engine then we could provide sanity checks for network programming, similar to the sanity checks given by the type systems in ML and Haskell.

References

Abadi, M., & Fournet, C. (2001). Mobile values, new names, and secure communication. *SIGPLAN Not.*, *36*(3), 104–115. http://doi.acm.org/10.1145/373243.360213

Caires, L., Pfenning, F., & Toninho, B. (2012). Towards concurrent type theory.

Church, A. (1985). *The Calculi of Lambda Conversion. (AM-6) (Annals of Mathematics Studies)*. Princeton, NJ, USA: Princeton University Press.

Curry, H. (1934). Functionality in combinatorial logic. In *Proceedings of National Academy of Sciences*, vol. 20, (pp. 584–590).

Fischer, M. J., & Merritt, M. (2003). Appraising two decades of distributed computing theory research. *Distrib. Comput.*, *16*(2-3), 239–247. http://dx.doi.org/10.1007/s00446-003-0096-6

Gentzen, G. (1935). Untersuchungen ber das logische schlieen. i. *Mathematische Zeitschrift*, *39*(1), 176–210. http://dx.doi.org/10.1007/BF01201353

Girard, J.-Y. (1995). logic: its syntax and semantics. In *Advances in Linear Logic*, (pp. 1–42). Cambridge University Press.

Hickey, R. (2013). Clojure core.async channels. http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall.

Hudak, P., Hughes, J., Peyton Jones, S., & Wadler, P. (2007). A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, (pp. 12–1–12–55). New York, NY, USA: ACM. http://doi.acm.org/10.1145/1238844.1238856

Jones, S. P. (2001). Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering theories of software construction*, (pp. 47–96). Press.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, *3*(4), 184–195. http://doi.acm.org/10.1145/367177.367199

Milner, R., Parrow, J., & Walker, D. (1992). A calculus of mobile processes, i. *Inf. Comput.*, *100*(1), 1–40. http://dx.doi.org/10.1016/0890-5401(92)90008-4

Milner, R., Tofte, M., & Macqueen, D. (1997). *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press.

Peyton Jones, S. L., & Wadler, P. (1993). Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, (pp. 71–84). New York, NY, USA: ACM. http://doi.acm.org/10.1145/158511.158524

Pfenning, F. (2012). 15-816 linear logic. University Lecture Series.

Pike, R., & Gerrand, A. (2013). Concurrency is not parallelism. http://blog.golang.org/concurrency-is-not-parallelism

Toninho, B., Caires, L., & Pfenning, F. (2013). Higher-order processes, functions, and sessions: A monadic integration. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, (pp. 350–369). Berlin, Heidelberg: Springer-Verlag. http://dx.doi.org/10.1007/978-3-642-37036-6_20

Toninho, B., Caires, L., & Pfenning, F. (2014). Corecursion in session-typed processes.

Wadler, P. (1993). A taste of linear logic. In A. Borzyszkowski, & S. Sokoowski (Eds.), *Mathematical Foundations of Computer Science 1993*, vol. 711 of *Lecture Notes in Computer Science*, (pp. 185–210). Springer Berlin Heidelberg. http://dx.doi.org/10.1007/3-540-57182-5_12

Wadler, P. (2006). Faith, evolution, and programming languages: from haskell to java to links. OOPSLA Lecture. http://homepages.inf.ed.ac.uk/wadler/topics/gj.html

Wadler, P. (2012). Propositions as sessions. *SIGPLAN Not.*, *47*(9), 273–286. http://doi.acm.org/10.1145/2398856.2364568

Wikipedia (2014). Wikipedia, the free encyclopedia. [Online; accessed 25-April-2012]. http://en.wikipedia.org/