The π-calculus

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

William Henderson

May 2008

Approved for the Division
(Mathematics)

James Fix

Table of Contents

Acknowledgments

I want to thank my hamsters, Boris Becker, and this bottle of Merlot.

A Word About Formatting

One of the pedagogical aims of this thesis is to be as clear and reader-assistive as possible during its presentation. To that end, I have taken several liberties in the formatting of this thesis. Figures, equations, definitions, theorems and examples all share one numbering scheme in hopes that it will make them easier to locate. Where *definitions* appear, they are clad in italics as well as in a margin box to make them easier to find. Equation numbers also appear in margin boxes. When definitions or equations are referred to later in the text, an assistive link will appear in the margin to avoid index-fingering. To this end, the wired reader will note the presence within the backcover of a searchable pdf version of the text, wherein all references are hyper-textual (clickable). The included CD also contains all the code referenced in this thesis.

definitions

definitions
↪ page iii

Abstract

In this thesis I will blow your mind, and possibly (hopefully?) my own in the process.

1 Introduction

Writing distributed systems are often held up as being some of the toughest problems that software engineers face today. Managing threads, locks and synchronization, avoiding deadlocks and livelocks and dealing with failure and latency all contribute to distributed programming nightmarish reputation. In the meantime, we steadily progress towards a future where all computation is distributed, where raw computing power is nothing more than a service that can be consumed like energy, and where huge systems sprawl across machines and networks to do their bidding.

Perhaps there is a better way. Many computer scientists are now reevaluating the tools they use – namely the languages – in hopes that there may be a way to cut the gordian knot by making very low level changes. The hope is that there is a breed of languages that offer a much more natural way to perform distributed computing, that do not have all the high-level nightmares that currently plague us. The aim of this thesis is to explore what such a language might look like, and perhaps offer some concrete features with which we might judge the overall value of a candidate language.

For several decades theoretical computer scientists have been studying a branch of math known as *processes algebras*. The hope is to discover an algebraic model for computation that might eventually be the foundation for a new wave of programming languages that greatly ease the difficulty of distributed programming by considering computation via its natural distributed unit - the process. A process can be a task of any size, but the important thing about it is that it shares no state with the processes in its environment. Instead, processes interact via message passing – sending data back-and-forth via named channels. The idea is that my modeling computation in such terms, we might discover natural ways to solve distributed problems and not have to worry about the many headaches that would otherwise arise at a very high level in a language whose primitives are not built for a distributed setting.

One of the most successful and influential process algebras to arise has be the π-calculus, first given by Robin Milner, Joachim Parrow and David Walker. It improved upon the process algebras that came before it by allowing processes a kind of mobility within their environments, which in turn allows for much more dynamic and powerful systems. We will give a formal presentation of the π-calculus, as a sort of starting point for our discussion of process algebra and the various languages they have inspired. In addition, the presentation will give us some valuable tools and vocabulary for evaluating these languages.

There has been a mushroom of process algebra that have sprung up around the π-

calculus, evidence of its fertility but also of the uncertainty of what the right system is. We will look at a few of these variations at a somewhat higher level than the π-calculus, and attempt to pull out some of the trends and major debates and clarify how they relate to the needs of distributed systems.

We will also look at a number of implementation of these languages, and use some example systems as a litmus test for now naturally they lend themselves to solving the problems.

2
The π-Calculus

In the 1930's, Alonzo Church and Stephen Kleene introduced the λ-calculus as an algebraic model for describing computation. In the λ-calculus, computation is the invocation of functions. In the 1980's, Robin Milner introduced his Calculus of Communicating Systems (CCS), in which computation is modeled as communication via message passing. Because these channels can be shared and used an arbitrary number of times, there is not a concrete naming system for a chunk of computation the way there is in a function. Instead, in the CCS any bit of functionality can be referred to as a *process*, with no specification of the granularity. Where we might think of a program in the λ-calculus as a bunch of functions invoking one another, in the the CCS we say that a *system* is composed of many communicating processes. It was this CCS that later became the basis for the π-calculus.

Although it can be defined in other ways, one of the ways of giving a process's *location* is in terms of the communication channels which can be used to access it. Since processes are the units populating the space of a CCS system, it is natural to think of a process's location in terms of the processes which are 'near' it - those it can communicate with via named channels. In this case, changing the channel topology of a system changes the locations of its component processes. In the CCS, channel topology is static, whereas the π-calculus allows communication channels to be dynamically established and relinquished between processes. This gives a kind of *mobility* of processes, which vastly expands the capabilities of interaction in a system.

In this chapter, we will give the syntax of the *asynchronous* π-calculus and a discussion of its features.[1] Following this, we will introduce a notion of equivalence of terms in the language. Finally, we will give some semantic reduction rules that provide computational behavior via steps of reduction.

2.1 Syntax

The terms of the π-calculus operate on a space of *identifiers* $v, w, ...$ which for the time being will consist of names $n, m, ...$ for communication channels, and variables $x, y, ...$ which, as we will see below, can refer to channels or to recursive process bodies.

identifiers

[1] Several versions of the π-calculus have been proposed. We will loosely follow the recent presentation given in [Hen07].

	Process terms	
$R, S, B :=$	$R_1 \mid R_2$	Composition
	$n!\langle V\rangle$	Send
	$n?(X).R$	Receive
	$\text{new}(n).R$	Restriction
	if $v_1 = v_2$ then R_1 else R_2	Matching
	rec $x.B$	Recursion
	stop	Termination
	Systems	
$\mathbb{M}, \mathbb{N} :=$	$\text{new}(c_1, ..., c_n).R_1 \mid ... \mid R_m$	$n, m > 0$

Figure 2.1.1: Terms in the asynchronous π-calculus

Given two or more processes, we can compose them using the $\mid$ operator, which means that the composed processes will be executed concurrently.

We denote the sending of a tuple V over a channel n by $n!\langle V\rangle$. Here V is a tuple of identifiers in the form $V = (v_1, ..., v_k) : k \geq 0$. We say that V has *arity* k. In the case $n = 0$ nothing is being transmitted; the communication acts as a *handshake* or signal. We will denote this case by $n!\langle\rangle$. It is a feature of the language that sending is not a blocking operation – that is, the process will not wait or depend on the value actually being received or operated on by another process, but will simply terminate after sending the value. Non-blocking sends make our π-calculus an *asynchronous* language - processes that send values do not wait to sync operation with the processes that consume the values. However, we will show in Example 2.1.9 that synchronous behavior can still be modeled in our language.

arity

handshake

The term $n?(X).R$ describes a process waiting to receive a tuple along k before continuing with R. Here X is a *pattern*, or a tuple of variables of arity k. Patterns allow us to decompose the transmitted tuple into its component values by naming them with $x_1, ..., x_k$, which can be referred to in R. Thus, in the term

patterns

$$c!\langle n_1, n_2, n_3\rangle \mid c?(x_1, x_2, x_3).R, \tag{2.1.2}$$

the names n_1, n_2, n_3 are received via c and bound via matching to x_1, x_2, x_3 (the use of x_1, x_2, x_3 is limited to being within the process R). Similarly to sending, the case of arity 0 is denoted $n?().R$ – here R will not happen until a handshake is received on n. Notice that in contrast to the case of sending, receiving is a *guarded* operation – that is, the process R will not execute until X has been received via n. For example, the term

guarded

$$c?().stop \tag{2.1.3}$$

represents a 'listener' process that simply consumes the value waiting on input channel c. The term *stop* describes a process that does nothing but halt.

The term $\text{new}(n).R$ describes a process in which a new channel n is created, limited to being expressed in the process R (we say the *scope* of n is *restricted* to R).

scope

When creating multiple channel restrictions of the form $\text{new}(n).(\text{new}(m).R)$, we will abbreviate using $\text{new}(n, m).R$. It is important to note that n *can* be used outside of R if it is sent and then received by some outside process. This feature is known as *scope extrusion*, and is the underpinning for the dynamic communication topologies introduced in the π-calculus. For example, in the term

scope extrusion

$$\text{new}(n).(c!\langle n\rangle) \mid c?(a).a!\langle\rangle, \tag{2.1.4}$$

n's scope is just the left half of the term. However, n is sent over c and is received (and bound as a) in the right hand of term. When this happens, the channel n will be able to be referred to outside of its initial scope. We will give a precise account of how this happens in our reduction rules and Example 2.3.2 below.

Simple value-matching is available to provide conditional behavior. In the following term, the value received on c is checked; if it matches a then a handshake is sent along a (which is referred to by x), otherwise the process terminates.

$$c?(x).(\text{if } x = a \text{ then } x!\langle\rangle \text{ else } \mathit{stop}) \tag{2.1.5}$$

Recursion is built into the language using the syntax $\text{rec } x.B$. The process $\text{rec } x.B$ itself is referred to by the variable x, which is used somewhere in B to express a recursive call. For example, consider the recursive responder term below, which receives a channel x_1 via c. It then sends a handshake response on x_1, while another responder process is run in parallel.

$$\text{rec } x.c?(x_1).(x_1!\langle\rangle \mid x) \tag{2.1.6}$$

We call a collection of parallel processes communicating over shared channels a *system*. Note that a system is itself a process. Our choice to define it is somewhat auxiliary to the π-calculus itself, but it is helpful to understanding the way that behavior can be modeled using processes as atoms of a system.

Example 2.1.7 Many presentations (including our own in the next chapter) of the π-calculus involve a choice or summation operator as in $P+Q$ for two processes terms P and Q. The meaning is essentially that either P or Q can be non-deterministically executed, and the other process will not be. However, we can model the same behavior without defining a choice operator:

$$c!\langle\rangle \mid c?().P_1 \mid c?().P_2 \tag{2.1.8}$$

In the above process, either Q or P (but not both) could be executed. This is due to the asynchronous behavior of our language – sending an empty signal along c, we cannot control which process consumes the signal (or even if it will be consumed at all). Thus, even without the choice operator our calculus is non-deterministic.

We can build systems where only one process will send a value along some channel c and only one will receive along. However, suppose we took our system and used it in building a larger system. Then we could have other processes that compete in sending or receiving along c and make our system non-deterministic. Nevertheless, we will see in the next example that we can at least learn *that* a value was consumed, although we can't say by *whom*. ■

Example 2.1.9 Perhaps we are not happy with this asynchronous transmission behavior. Surely we'd like to have blocking sends sometimes, or be able to guarantee that a value is received. However, it can be shown that our asynchronous languages is more general than a synchronous version. To see why, consider the following system:

$$F_1(c) \Leftarrow \text{rec } z.(\text{new}(ack).(c!\langle ack\rangle \mid ack?().(R \mid z))) \tag{2.1.10}$$
$$F_2(c) \Leftarrow \text{rec } q.(c?(ack).(ack!\langle\rangle \mid q)) \tag{2.1.11}$$
$$Sys_1 \Leftarrow \text{new}(d).(F_1(d) \mid F_2(d)) \tag{2.1.12}$$

interface

First, we use $\Leftarrow$ with $F_1(c)$ to mean that the process term on the right with the channel c will be denoted by $F_1(c)$ which we will call the *interface* of a system or process. Hence, $F_1(d)$ would represent $F_1(c)$, with the occurrences of c replaced with d. Both F_1 and F_2 have (infinite) recursive behavior, and we can think of Sys_1 as a term that 'kicks off' both of them after creating a shared channel for them to communicate on.

In an iteration of F_1, a new channel ack is created for acknowledgement. This channel is sent along c and then F_1 waits for input on ack before executing some term R and calling another iteration. F_2 receives ack along c and then uses ack to send an empty signal to F_1 that it has received input on c.

Of course, this is a silly example since the only thing being communicated is the acknowledgement channel itself – we might imagine a more complex system where F_1 sends more input along c and waits to make sure F_2 receives it. Note that the channel ack is only used *once* – we want to ensure that the acknowledgement that F_1 receives is definitely for *that* instance of communication that it just initiated. This allows us to guarantee that F_2 has executed before F_1 continues with R. ■

2.2 Structural Equivalence

A natural question at this point is, given two π-calculus terms, how can we determine if they are equivalent? Intuitively, we want them to be equivalent if they *act* the same, but actually defining this equivalence relation can be a bit subtle. First, we will look at substitution and give some rules for when we can safely interchange identifiers without creating a different term. Following this, we will develop the notion of *contexts* and then use it to build an equivalence relation among processes.

2.2.1 Identifier Substitution and α-equivalence

As a first step in our notion of equivalence, we might assert that the way things are named shouldn't change how they act. Of course, this doesn't mean we can start interchanging symbols with carefree abandon – only those identifiers whose use is *bound* in the process. There are three ways that identifiers can become bound in the π-calculus. First, a name n is bound in the restriction operator, $\text{new}(n).R$. Identifiers x_i can be bound when they are part of the pattern X matched in the receive expression $c?(X).R$. Finally, x can be bound in rec $x.B$.

bound identifiers

identifiers
↪ page 3

We denote the set of bound identifiers in a term R by $bi(R)$; all those which are not bound we call *free*, denoting them $fi(R)$. We call a term with no free variables *closed*. Since such a term is self-contained in that it doesn't need any outside context to make sense, we will finally give our precise definition for *processes* as terms that are closed.

free identifiers

closed terms

processes

In term (2.1.6) above, x is a bound recursive variable, while x_1 is bound by the receive operator. Now consider the term

$$c!\langle n\rangle \mid \text{new}(n).(\text{rec } x.c?(x_1).(x_1!\langle\rangle \mid x)) \tag{2.2.1}$$

Here x_1, x are bound as before. However, the use of n being sent on c is *not* bound, since it occurs outside the scope of the scope restriction operator. This is clear in the following term, which is equivalent (see Example 2.2.8 for justification).

$$a?(x_1, x_2).x_1!\langle\rangle \mid \text{rec } x.(a!\langle n, m\rangle \mid x) \tag{2.2.2}$$

We denote the substitution of values in V for free identifiers X in R by $R[\![V/X]\!]$. Of course, during the course of a substitution, we might inadvertently 'capture' a bound term – suppose for example we wanted to perform the substitution $P_1(c)[\![n/c]\!]$ where

$$P_1(c) \Leftarrow \text{new}(n).(n!\langle x\rangle \mid c!\langle y\rangle) \tag{2.2.3}$$

This would make the free identifier c into a bound identifier n. We can avoid this by first replacing n a new variable that is 'fresh' – ie it does not occur elsewhere in the term:

$$P_1^{'}(c) \Leftarrow \text{new}(n').(n'!\langle x\rangle \mid c!\langle y\rangle) \tag{2.2.4}$$

We can now safely perform the substitution $P_1^{'}(c)[\![n/c]\!]$, yielding

$$\text{new}(n').(n'!\langle x\rangle \mid n!\langle y\rangle) \tag{2.2.5}$$

Obviously we want to say $P_1(c)$ and $P_1^{'}(c)$ are equivalent. In general when two terms are the same up to the names of bound identifiers, we say they are *α-equivalent*, and write $P_1(c) \equiv_\alpha P_1^{'}(c)$. We will intend from now on that a term represents its entire α-equivalency class, and thus will not explicitly specify α-equivalency in the equivalence relation below.

α-equivalency

2.2.2 Contexts and Equivalence

Perhaps the next idea we might have for building our equivalence relation is that equivalent processes should act the same when dropped into any larger system. Let us define more precisely what we mean by 'dropping in' a process.

Definition 2.2.6 (Context) A context $\mathbb{C}$ is given by:

$$\mathbb{C} := \begin{cases} [\,] & \\ \mathcal{C} \mid Q \text{ or } Q \mid \mathcal{C} & \text{for any process } Q, \text{ context } \mathcal{C} \\ \text{new}(n).\mathcal{C} & \text{for any name } n, \text{ context } \mathcal{C}. \end{cases}$$

$\mathbb{C}[Q]$ denotes the result of replacing the placeholder $[\,]$ in the context $\mathbb{C}$ with the process Q.

Notice that with contexts, we do not pay any attention to whether a name in Q is bound in $\mathbb{C}$. Hence, unlike with substitution, free variables in Q can become bound in $\mathbb{C}[Q]$. We say that a relation $\sim$ between processes is *contextual* if $P \sim Q$ implies $\mathbb{C}[P] \sim \mathbb{C}[Q]$ for any context $\mathbb{C}$. We are now ready to define our notion of equivalency using contexts.

contextual

Definition 2.2.7 (Structural Equivalence) Structural Equivalence, denoted $\equiv$ is the smallest contextual equivalence relation that satisfies the following axioms:

$$\begin{aligned}
P \mid Q &\equiv Q \mid P && \text{(S-COMP-COMM)} \\
(P \mid Q) \mid R &\equiv P \mid (Q \mid R) && \text{(S-COMP-ASSOC)} \\
P \mid \text{stop} &\equiv P && \text{(S-COMP-ID)} \\
\text{new}(c).\text{stop} &\equiv \text{stop} && \text{(S-REST-ID)} \\
\text{new}(c).\text{new}(d).P &\equiv \text{new}(d).\text{new}(c).P && \text{(S-REST-COMM)} \\
\text{new}(c).(P \mid Q) &\equiv P \mid \text{new}(c).Q, \text{ if } c \notin fi(P) && \text{(S-REST-COMP)}
\end{aligned}$$

These axioms are simply a set of properties that we expect our syntax to obey. The first and second state that composition is commutative and associate. Thus, we will omit parenthesis around compositions when our meaning is clear. (S-COMP-ID) states that a terminated process can be eliminated from a composition. (S-REST-ID) states that a scope restriction operator can be eliminated when its scope is only a terminated process. (S-REST-COMP) states that scope ordering does not matter, justifying our shorthand $\text{new}(c,d).P$. It is the last of these axioms that is most important – it is the basis for scope extrusion, upon which the language's process mobility is based (as we shall demonstrate in Example 2.3.2 below).

scope extrusion
↪ page 5

Example 2.2.8 In our discussion of bound identifiers above, we asserted that the elimination of a scope restriction when none of the scoped identifiers occur in its scope. We can now show why this is permissible:

$$\begin{aligned}
& \text{new}(n,m).(a?(x_1,x_2).x_1!\langle\rangle) \mid \text{rec } x.(a!\langle n,m\rangle \mid x) & \\
\equiv\ & \text{new}(n,m).(a?(x_1,x_2).x_1!\langle\rangle \mid \mathit{stop}) \mid \text{rec } x.(a!\langle n,m\rangle \mid x) && \text{(S-COMP-ID)} \\
\equiv\ & a?(x_1,x_2).x_1!\langle\rangle \mid \text{new}(n,m).(\mathit{stop}) \mid \text{rec } x.(a!\langle n,m\rangle \mid x) && \text{(S-REST-COMP)} \\
\equiv\ & a?(x_1,x_2).x_1!\langle\rangle \mid \mathit{stop} \mid \text{rec } x.(a!\langle n,m\rangle \mid x) && \text{(S-REST-ID)} \\
\equiv\ & a?(x_1,x_2).x_1!\langle\rangle \mid \text{rec } x.(a!\langle n,m\rangle \mid x) && \text{(S-COMP-ID)}
\end{aligned}$$

■

2.3 Reduction Semantics

We also want to give some *semantic* properties that a process in our language should possess. Here we do not define a relation of equivalence but rather one that allows a process to internally evolve through a number of computation steps.

Definition 2.3.1 (Reduction) The *reduction relation* $\longrightarrow$ is the smallest contextual relation that satisfies the following rules:

$$c!\langle V\rangle \mid c?(X).R \longrightarrow R[\![V/X]\!] \quad \text{(R-COMM)}$$
$$\text{rec } x.B \longrightarrow B[\![\text{rec } x.B/x]\!] \quad \text{(R-REP)}$$
$$\text{if } v = v \text{ then } P \text{ else } Q \longrightarrow P \quad \text{(R-EQ)}$$
$$\text{if } v_1 = v_2 \text{ then } P \text{ else } Q \longrightarrow Q \quad (\text{where } v_1 \neq v_2) \quad \text{(R-NEQ)}$$
$$\frac{P \equiv P', P \longrightarrow Q, Q \equiv Q'}{P' \longrightarrow Q'} \quad \text{(R-STRUC)}$$

We use the notation $P \cdots\!\longrightarrow Q$ when an arbitrary number of these rules have been applied in reducing P to Q.

The first of these allows a computation step for the transmission of values from one process to another. (R-REP) allows us to unravel a recursive expression into iterations of itself. (R-MATCH) enables a computational step for value-matching. Finally, (R-STRUCT) says that a reduction is defined up to structural equivalence.

Example 2.3.2 We will give a demonstration of how scope extrusion is defined using the rules and axioms of reduction and structural equivalence. Consider the expression

$$d?(X).X!\langle\rangle \mid \text{new}(c).(d!\langle c\rangle \mid c?().stop) \tag{2.3.3}$$

We can use (S-REST-COMP) to bring the restriction to the outside, giving:

$$\text{new}(c).(d?(X).X!\langle\rangle \mid d!\langle c\rangle \mid c?().stop) \tag{2.3.4}$$

Thanks to the reduction relation's contextuality, we can use (R-COMM) inside of the restriction operator and apply a substitution. Using this in conjunction with the above structural equality and (R-STRUCT), we get:

$$\text{new}(c).(c!\langle\rangle \mid c?().stop) \tag{2.3.5}$$

Finally, we can apply (R-STRUCT) again, so the process simply reduces to stop.

Now let $P(d)$ to be the right half of our original process, and $\mathbb{C}$ to be the remaining context:

$$P(d) \Leftarrow \text{new}(c).(d!\langle c\rangle \mid c?().stop) \tag{2.3.6}$$

$$\mathbb{C} = d?(X).X!\langle\rangle \mid [\,] \tag{2.3.7}$$

Then we have shown that if we drop $P(d)$ into the context $\mathbb{C}$ (forming $\mathbb{C}[P(d)]$), it will establish a new channel c and extend its scope using d. Hence it is (S-REST-COMP) which allows our system's communication topology to change dynamically via scope extrusion. ■

Example 2.3.8 Suppose we wanted to model a simple memory cell with channels for getting and setting its value. It turns out that we can simulate this just using message passing. The problem that we have to work around is that receiving a identifier from a channel removes that value from the channel – it cannot be retrieved again. Thus, we need a way to push it back in for the next time. Consider the following system:

$$\begin{aligned} Cell(get, set) \Leftarrow\ & \text{new}(c).(c!\langle init\rangle \\ & \quad \mid \text{rec } g.(get?(b).c?(y).(c!\langle y\rangle \mid b!\langle y\rangle \mid g)) \\ & \quad \mid \text{rec } s.(set?(x,b).c?(y).(c!\langle x\rangle \mid b!\langle x\rangle \mid s))) \end{aligned} \tag{2.3.9}$$

In the cell, we first create a new channel c that will be used to 'store' the value. We then initialize c with the value *init*. The following line has a recursive listener process than, when given a channel via *get*, gets the value from c and sends it back via the supplied channel. In parallel, it sends the value back into c so it can be had again. The next line is similar except that two values are supplied via *set* - a new value and a channel. The old value is pulled from c and simply discarded (ie not used), whereas the new value x is pushed to the c this time. x is also sent via b as an acknowledgment that the cell's new value has been set.

Also notice that the system has the free channels get_c, set_c given in the interface. When some larger system binds them and uses them to interact with the components inside $Cell(get_c, set_c)$ we say that the system *instantiates* $Cell(get_c, set_c)$. In the following example we assume some larger system uses *printer* to actually print the message.

interface
↪ page 6

instantiate

$$\begin{aligned} & Echo(printer) \Leftarrow \\ & \quad \text{new}(get_1, set_1).(Cell\langle get_1, set_1\rangle \\ & \qquad \mid \text{new}(a).(set_1!\langle \text{“}hello,\ world\text{”}, a\rangle \\ & \qquad\quad \mid a?(x).(get_1!\langle a\rangle \mid a?(y).printer!\langle y\rangle))) \end{aligned} \tag{2.3.10}$$

The system *Echo* creates new channels to interact with the instance of memory cell it spawns, while in parallel it puts a message in the cell, then (order is ensured using by waiting for the response via a) getting the value from the cell and sending it to the *printer* channel.

To see why this produces the behavior we expect, first observe scope restrictors

for a and c can be moved out using $(S - REST - COMP)$:

$$\begin{aligned}\text{new}(get_1, set_1, a, c).\Big(&c!\langle init\rangle \mid \text{rec } g.(get?(b).c?(y).(c!\langle y\rangle \mid b!\langle y\rangle \mid g))\\ &\mid \text{rec } s.(set?(x,b).c?(y).(c!\langle x\rangle \mid b!\langle x\rangle \mid s))\\ &\mid set_1!\langle \text{“}hello,\ world\text{”}, a\rangle\\ &\mid a?(x).(get_1!\langle a\rangle \mid a?(y).printer!\langle y\rangle)\Big)\end{aligned} \tag{2.3.11}$$

Two applications of (R-REP) allow us to 'pull out' a recursive call:

$$\begin{aligned}&\text{new}(get_1, set_1, a, c).\\ &\Big(c!\langle init\rangle \mid get?(b).c?(y).(c!\langle y\rangle \mid b!\langle y\rangle) \mid \text{rec } g.(get?(b).c?(y).(c!\langle y\rangle \mid b!\langle y\rangle \mid g))\\ &\mid set?(x,b).c?(y).(c!\langle x\rangle \mid b!\langle x\rangle) \mid \text{rec } s.(set?(x,b).c?(y).(c!\langle x\rangle \mid b!\langle x\rangle \mid s))\\ &\mid set_1!\langle \text{“}hello,\ world\text{”}, a\rangle\\ &\mid a?(x).(get_1!\langle a\rangle \mid a?(y).printer!\langle y\rangle)\Big)\end{aligned} \tag{2.3.12}$$

Now we can apply (R-COMM) to the memory cell's setter...

$$\begin{aligned}&\text{new}(get_1, set_1, a, c).\\ &\Big(c!\langle init\rangle \mid get?(b).c?(y).(c!\langle y\rangle \mid b!\langle y\rangle) \mid \text{rec } g.(get?(b).c?(y).(c!\langle y\rangle \mid b!\langle y\rangle \mid g))\\ &\mid c?(y).(c!\langle \text{“}hello,\ world\text{”}\rangle \mid a!\langle \text{“}hello,\ world\text{”}\rangle\\ &\qquad \mid \text{rec } s.(set?(x,b).c?(y).(c!\langle x\rangle \mid b!\langle x\rangle \mid s)))\\ &\mid a?(x).(get_1!\langle a\rangle \mid a?(y).printer!\langle y\rangle)\Big)\end{aligned} \tag{2.3.13}$$

...which lets us strip out the initializer, and in turn the acknowledgement on a of setting "hello, world" using (R-COMM),

$$\begin{aligned}&\text{new}(get_1, set_1, a, c).\\ &\Big(get?(b).c?(y).(c!\langle y\rangle \mid b!\langle y\rangle) \mid \text{rec } g.(get?(b).c?(y).(c!\langle y\rangle \mid b!\langle y\rangle \mid g))\\ &\mid c!\langle \text{“}hello,\ world\text{”}\rangle \mid \text{rec } s.(set?(x,b).c?(y).(c!\langle x\rangle \mid b!\langle x\rangle \mid s))\\ &\mid get_1!\langle a\rangle \mid a?(y).printer!\langle y\rangle\Big)\end{aligned} \tag{2.3.14}$$

We can now use (R-COMM) on the memory cell's getter and also on its use of c to get the "hello, world" state:

$$\begin{aligned}&\text{new}(get_1, set_1, a, c).\\ &\Big(c!\langle \text{“}hello,\ world\text{”}\rangle \mid a!\langle \text{“}hello,\ world\text{”}\rangle \mid \text{rec } g.(get?(b).c?(y).(c!\langle y\rangle \mid b!\langle y\rangle \mid g))\\ &\mid \text{rec } s.(set?(x,b).c?(y).(c!\langle x\rangle \mid b!\langle x\rangle \mid s))\\ &\mid a?(y).printer!\langle y\rangle\Big)\end{aligned} \tag{2.3.15}$$

A final application of (R-COMM) to a gives us

$$\begin{aligned}&\text{new}(get_1, set_1, a, c).\\&\Big(c!\langle \text{“}hello,\ world\text{”}\rangle \mid \text{rec } g.(get?(b).c?(y).(c!\langle y\rangle \mid b!\langle y\rangle \mid g))\\&\mid \text{rec } s.(set?(x,b).c?(y).(c!\langle x\rangle \mid b!\langle x\rangle \mid s))\\&\mid printer!\langle \text{“}hello,\ world\text{”}\rangle\Big)\end{aligned} \tag{2.3.16}$$

which is essentially equivalent to initializing a memory cell with “hello, world” and also sending the message to *printer*. ■

2.4 Action Semantics

Our description of process behaviors so far has been limited to talking about the internal computational steps through which it might evolve. Now, we want to give a more general description of how a process might evolve in the context of a larger system. In such a system, a process can be said to either send or receive values along channels it shares with the system. To describe these abilities, we will use the notion of a *labelled transition system*, or *lts*.

Definition 2.4.1 (Labelled Transition System) A *labelled transition sytem* $\mathcal{L}$ is a tuple $(\mathcal{S}, \mathcal{A})$ where $\mathcal{S}$ is a set of processes and $\mathcal{A}$ is a set of labels called *actions*. Furthermore, for each action a, there is a binary relation:

$$R_a \subseteq \mathcal{S} \times \mathcal{S}$$

To denote that $\langle P, Q\rangle \in R_a$, we will use the notation $P \xrightarrow{a} Q$.

Hence, the transition $P \xrightarrow{a} Q$ indicates that there is an action under which the process P becomes Q. We will refer to Q is the *residual* of P after a.

residual

There are three types of actions that may cause a process to evolve. Note that we will use α to refer to an action of arbitrary type. First, the process might receive a value. That is, a process P can be said to be capable of the transition $P \xrightarrow{c?X} Q$, which is to say P can receive X along c resulting in the residual Q.

The second type of action available is sending. Here we need to be a bit more careful. In the case of receiving, the received name is always bound to a new name X - so we needn't worry about issues of scoping. In sending, however, we might be transmitting either a free or a bound name (or a mix of them in a tuple). In the latter case, we need to take account of the fact that the scope of the name is being extruded to whatever process receives the name. We denote the set of bound names in the send action by (B), and we say that this set of names is *exported* by the process. Hence, the transition $P \xrightarrow{(B)c!V} Q$ represents that P can send V over c, exporting (B) and resulting in Q. For example,

scope extrusion
↪ page 5

exported

$$\text{new}(a).c!\langle a\rangle \mid Q \xrightarrow{(b)c!a} Q$$

Finally, we have a third action available for *internal evolution*. This is caused by some internal evolution in P like those described in Section 2.3. We use τ denote such an internal evolution step. Thus, we say $P \xrightarrow{\tau} Q$ if P is able evolve into Q by performing a reduction step without any external actions. For example (thanks to (R-COMM)),

internal evolution

$$c!\langle a\rangle \mid c?(x).x!\langle\rangle \xrightarrow{\tau} c!\langle\rangle$$

Using these actions, we can give a set of rules describing the behavior of a π-calculusprocesses in an arbitrary context. Hence, we now formally define the action relation under these rules:

Jim:
I wonder if these couldn't be simplified by allowing a transition that happens over the reduction semantics?
i can't find anyone in the literature who does this but it seems like the best approach...

Definition 2.4.2 (Action) The *action relation* $\longrightarrow$ is the smallest relation between processes that satisfy the following rules:

$$c?(X).R \xrightarrow{c?X} \mathrm{R}[\![V/X]\!] \qquad \text{(A-IN)}$$

$$c!\langle V\rangle \xrightarrow{c!V} stop \qquad \text{(A-OUT)}$$

$$\mathrm{rec}\ x.R \xrightarrow{\tau} R[\![\mathrm{rec}\ x.R/x]\!] \qquad \text{(A-REP)}$$

$$\text{if } v = v \text{ then } P \text{ else } Q \xrightarrow{\tau} P \qquad \text{(A-EQ)}$$

$$\text{if } v_1 = v_2 \text{ then } P \text{ else } Q \xrightarrow{\tau} Q \qquad v_1 \neq v_2 \qquad \text{(A-NEQ)}$$

$$\frac{P \xrightarrow{(B)c!V} P'}{\mathrm{new}(n).P \xrightarrow{(n,B)c!V} P'} \qquad \text{(A-OPEN)}$$

$$\frac{P \xrightarrow{c?X} P',\ Q \xrightarrow{(B)c!V} Q'}{P \mid Q \xrightarrow{\tau} \mathrm{new}(B).(P' \mid Q')} \qquad (B) \cap fn(Q) = \emptyset \qquad \text{(A-COMM)}$$

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad bn(\alpha) \cap fn(Q) = \emptyset \qquad \text{(A-COMP)}$$

$$\frac{P \xrightarrow{\alpha} P'}{\mathrm{new}(b).P \xrightarrow{\alpha} \mathrm{new}(b).P'} \qquad b \notin n(\alpha) \qquad \text{(A-REST)}$$

3 Synchronicity and Expressiveness

In Section ??, we introduced a version of the π-calculus which we said was made *asynchronous* by its use of non-blocking send operations. Rather than allowing for an operation to happen after a sent value has been received by some other process, a sending process simply ends, hoping someone will eventually consume the sent message. In Example 2.1.9 we gave a straightforward method of simulating synchronous sending the asynchronous π-calculus. We also briefly discussed the non-deterministic choice operator, absent in our calculus, and in Example 2.1.7 gave a method of simulating that as well.

The original calculus given by Milner, Parrow and Walker modeled synchronous message passing, and included the two operators – summation and synchronous sending – that we omitted in our asynchronous π-calculus. When combined, these two operators yield a unique behavior. We will give a presentation of the synchronous π-calculusin Section 3.1, along with a few examples of its power and use.

guarded
↪ page 4

In the synchronous π-calculus, we have two guarded processes - sending and receiving. If we have a group of guarded processes joined by the summation operator, only one of those will be processes will be 'chosen' to be executed. The rest will simply terminate without having any effect. The power of the synchronous π-calculuscomes when we have another group of summed processes, running in parallel, something like:

$$c!\langle\rangle.P_1 + d_1!\langle\rangle.P_2 + d_2?().P_3 \mid c?().R_1 + d_3!\langle\rangle.R_2$$

In this case, assuming the above is not part of a larger system, we can actually know (despite the non-deterministic nature of the choice operator) which of the processes will be chosen to run. Because both sends and receives are guarded the P_k's and R_K's can only run when their respective transmission operations complete. Hence, only P_1 and R_1 will be allowed to run since sending on c is the only transmission with a matching reception on c. Now imagine a slightly more complicated system:

$$c!\langle\rangle.P_1 + d!\langle\rangle.P_2 \mid c?().R_1 + d?().R_2$$

Here we have *two* transmissions with matches, so we cannot make a guarantee about what processes are chosen. However, we *can* say that if P_1 executes on the left side, R_1 will be chosen on the right side. We can say the same for P_2 and R_2. This is the power of the synchronous π-calculus- that there is a silent, implicit sort of communication that can happen between groups of parallel processes when a non-deterministic choice is made among them.

A natural question arises at this point: whether we can represent this expressive communication power using only our asynchronous π-calculus. That is, using the simulations given in Examples 2.1.9 and 2.1.7 (or some similar but more complicated approach) can we fully capture the kind of behavior discussed above? In Section **??**, we will explore the surprising complexity of this question and some of its implication.

Jim: On second thought, I think I might want to leave the reader with this question, for now - to peak their curiosity, perhaps. Is that alright?

3.1 The Synchronous π-Calculus

Action Prefixes

$\pi :=$	$n!\langle V\rangle$	Send
	$n?(X)$	Receive
	τ	Internal Evolution

Process terms

$R, S, B :=$	$\sum_{i\in\{1,..,n\}} \pi_i.R_i$	Summation ($n \in \mathbb{Z}$)
	$R_1 \mid R_2$	Composition
	$\text{new}(n).R$	Restriction
	if $v_1 = v_2$ then R_1 else R_2	Matching
	rec $x.B$	Recursion
	stop	Termination

Figure 3.1.1: Terms in the synchronous π-calculus

$$\begin{aligned}
\tau.R + P &\longrightarrow R && \text{(R-TAU)}\\
c!\langle V\rangle.P + Q \mid c?(X).R + B &\longrightarrow (P \mid R)[\![V/X]\!] && \text{(R-SYNC)}\\
\text{rec } x.B &\longrightarrow B[\![\text{rec } x.B/x]\!] && \text{(R-REP)}\\
\text{if } v = v \text{ then } P \text{ else } Q &\longrightarrow P && \text{(R-EQ)}\\
\text{if } v_1 = v_2 \text{ then } P \text{ else } Q &\longrightarrow Q \quad (\text{where } v_1 \neq v_2) && \text{(R-NEQ)}\\
\frac{P \equiv P', P \longrightarrow Q, Q \equiv Q'}{P' \longrightarrow Q'} & && \text{(R-STRUC)}
\end{aligned}$$

Figure 3.1.2: Reduction rules for the synchronous π-calculus

Note the important difference between Figure 2.1.1 and Figure 3.1.1. First, we have grouped sending and receiving together as *action prefixes* (along with a new prefix for internal evolution). These prefixes are made available to the language via the

internal evolution ↪ page 13

summation operator.

The notation $\pi_i.R_i$ requires that the action π_i happen before the guarded process R_i will be executed. If R_a is executed in this way, then for all $j \neq a$, both the capabilities π_j and the execution of R_j are lost. In other words, the summation ensures that only one of n guarded processes will be executed. Which of these processes is picked depends on which action prefix capability is exercised first.

For the cases that $n = 1$ and $n = 2$, we will use the notation $\pi.R$ and $\pi_1.R_1 + \pi_2.R_2$, respectively. We add to our congruence relation given in Definition 2.2.7 that summation is commutative and associative.

The summation case of $n = 0$ is what is meant by our *stop* termination process. It is thus trivially true that:

$$\pi.R + 0 \equiv \pi.R$$

guarded
↪ page 4

Note also that in our action prefixes, we have made sending a guarded operation, which means that R will not execute until some other process receives V along n. Receiving is also guarded, as in the asynchronous version. In the case where π is some internal evolution τ, we simply mean that R can proceed after the process does something which we cannot observe. This internal behavior is captured in the new reduction rule (R-TAU), while (R-SYNC) expresses the external evolution of the summation operator.

Example 3.1.3 It is not hard to show that the synchronous π-calculus can model the asynchronous version. To see why, first note that asynchronous sending can be encoded simply by $n!\langle V\rangle.stop$. This we will abbreviate with the familiar notation $n!\langle V\rangle$. As we noted above, the summation notation allows for a single guarded process. If we limit ourselves to these single summations, disallow internal action prefixes, and limit all sending to be of the form $n!\langle V\rangle.stop$, then we have the asynchronous π-calculus. To see why the reduction semantics are compatible, note that (R-TAU) is not needed in the asynchronous calculus since we have excluded its operator. The following shows that (R-SYNC) is a more general form of (R-COMM) of Definition 2.3.1:

Jim: worth showing? somewhat trivial...

$$\begin{aligned}
c!\langle V\rangle.stop + stop \mid c?(X).R + stop &\longrightarrow (stop \mid R)[\![V/X]\!] && \text{(R-SYNC)}\\
c!\langle V\rangle.stop + stop \mid c?(X).R + stop &\equiv c!\langle V\rangle \mid c?(X).R && \text{(S-COMP-ID)}\\
(stop \mid R)[\![V/X]\!] &\equiv R[\![V/X]\!] && \text{(S-COMP-ID)}\\
c!\langle V\rangle \mid c?(X).R &\longrightarrow R[\![V/X]\!] && \text{(R-STRUC)}
\end{aligned}$$

■

Jim: this example will likely need to be extended to support the action semantics being added to the previous chapter. I wonder also if I am right in thinking bisimulation can be avoided entirely for this encoding?

3.2 Separation Results

When trying to compare the expressive power of different calculi, one good approach is to, as above, provide explicit encodings from one language to another[1]. We say that

[1]Note that Example 3.1.3 is a trivial example of such an encoding since your encodings are straight-forward enough to be equivalent under the structural equivalence. Most encodings require

we are trying to encode from *source language* into the terms of a *target language*, and if we succeed, we have shown that the target language is at least as expressive as the source. Hence, in example Example 3.1.3 we showed that the synchronous calculus is at least as expressive as the asynchronous calculus by giving an encoding from the asynchronous to the synchronous. We will also use the notation

source language

target language

$$[\![P]\!] \stackrel{def}{=} Q$$

to mean that P in the source language is encoded by Q in the target language.

To prove a separation result between languages, it is enough to show that there are problems that are not solvable in the source language that are not solvable in the target language. In [Pal03], Palamidessi uses the solvability of the leader election problem on symmetric networks to show that the synchronous π-calculus is strictly more expressive than the asynchronous version. Loosely, the leader election problem is the problem of having group of identifier (via integers, perhaps) processes agree on a 'leader' process identification in a finite amount of time. We say that these processes are a symmetric network if any two processes P_i, P_j are equivalent under structural equivalence and renaming of their identifiers. For example, consider the following symmetric network:

$$P_0 \mid P_1 \Leftarrow \; c_0!\langle\rangle.o!\langle 0\rangle + c_1?().o!\langle 1\rangle \mid c_1!\langle\rangle.o!\langle 1\rangle + c_0?().o!\langle 0\rangle$$

It should not be hard to see that this solves the leader election problem in the synchronous π-calculus by agreeing on a leader via the output channel o. It may be less obvious, but it is not possible to solve the leader election problem in a symmetric network of asynchronous π-calculus processes. This is a direct result of the lack of the choice operator: without it, the symmetric processes have no way to pick a leader non-deterministically without potentially disagreeing with one another. It is only through the implicit communication underlying the choice operator that synchronous processes are able to break out of their symmetry and agree on a leader.

Using these results, Palamidessi a set of useful requirements that formally separate the two calculi.

The first of those requirement is on *uniformity*, which means that:

uniformity

$$[\![\alpha(P)]\!] = \alpha([\![Q]\!]) \tag{3.2.1}$$
$$[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!] \tag{3.2.2}$$

(3.2.1) simply states α-renaming is not violated in the process of the encoding. (3.2.2) is related to the requirements of a distributed system. That is, parallel processes really should just map to parallel processes, with not top level manager process or the like to aide the encoding. This is how Palamidessi builds the requirement of a symmetric network into a language requirement.

Jim: When I try to refer to the label α-equivalency here, things break. any idea how to fix this without re-engineering my definitions?

TODO: where/how?

The other requirement is on *reasonability*. Reasonability to Palamidessi means

reasonability

the more advanced structure of *bisimulation* equivalences, which relies on setting up a notion of simulation and then proving that two processes simulate one another.

that the language can distinguish between two processes when their actions are different on a certain given channel. This essentially encapsulates the requirements of the leader election problem. That is, a electoral system would be one were actions on the output channel are the same and we want our target language to be capable of semantically distinguishing this from a non-electoral system (where actions on the output channel differ).

4

Synchronicity and Distributed Systems

The second question we might ask is about the implementation of a synchronous π-calculus. On distributed systems, which seem like the natural setting for such an implementation, we have only asynchronous available to us. Hence, depending on the answer to the first question, we may have to

However, whether the asynchronous calculus actually has the expressive power of its synchronous forebear is a matter of more complexity. In fact, we will see by straightforward counterexample (and by a rather less straightforward argument due to Palamidessi [Pal03]) that two are not truly equal. The implementation of synchronous calculi on distributed systems is a thorny issue. On the one hand, it is a useful construct in that it allows us to model many problems more naturally and easily. On the other, all communication in a distributed system is in fact asynchronous in nature, and so synchronous communication must be layered on top at some level of the implementation. It is notoriously difficult construct to implement synchronous communication efficiently without violating the requirements of a truly distributed system where there is shared memory or central control process. The issue becomes more difficult still with the Palamidessi's related requirement of symmetry, which means that a process's behavior does not depend on its location in the channel topology.

TODO: make a note about why the original example only covered input guarded choice

The creators of Pict, the Join-calculus, and other implementations based on the π-calculus all decided to have their primitives support only asynchronous communication, while synchronous communication is made available overtop of this via a library or higher-level language. This these greatly simplifies implementation, resulting in a cleaner, more efficient core language. The summation operator in particular is difficult and expensive to fully simulate. In the implementation of Pict, for example, David Turner notes [Tur96] that "the additional costs imposed by summation are unacceptable.". Turner goes on to say that essential uses of summation are infrequent in practice.

How can we reconcile this with Palamidessi's result, which indicates that there are important problems that cannot be solved without the full generality of the synchronous calculus? Is such a calculus even implementable at any cost on distributed systems? We will see that relaxing any of Palamidessi's assumptions enables a full encoding. Some of such encodings derive from the classic distributed solutions to the problem of synchronous communication but strain important assumptions about symmetry in ways that we may not be comfortable allowing. Palamidessi herself gives as important probabilistic encoding [PH01] which does not break symmetry.

4.1 Symmetry in Practice

Speaking in an interview on developing the π-calculus, Robin Milner notes [Ber03]:

> That was to me the challenge: picking communication primitives which could be understood at a reasonably high level as well as in the way these systems are implemented at a very low level...There's a subtle change from the Turing-like question of what are the fundamental, smallest sets of primitives that you can find to understand computation...as we move towards mobility... we are in a terrific tension between (a) finding a small set of primitives and (b) modeling the real world accurately.

This tension is quite evident in the efforts of process algebraists to find the 'right' calculus for modeling distributed systems. While the synchronous π-calculus more elegantly and (given certain assumptions) completely expresses distributed systems, actual implementation must commit to asynchronous communication as their primitives. Which we choose as a model depends in part on our goals. In any case, it is evident that by limiting ourselves to smaller calculi, many useful new concepts and structures arise in order to solve the problems posed by asynchronous communication. While these structures might not belong in the 'smallest set of primitives', they are useful for bringing the power of the π-calculus to a model that more closely resembles the implementation of distributed systems.

This section will go on to talk about on Nestmann and other implementation-driven encodings, how much they violate uniformity/reasonableness and whether that has any means for the API as a good model for implementable distributed languages.

List of Figures

[Ber03] Martin Berger. An interview with robin milner, September 2003.

[Cas02] I. Castellani, editor. *Models of Distribution and Mobility: State of the Art*. MIKADO Global Computer Project, August 2002.

[Hen07] Matthew Hennessy. *A Distributed Pi-Calculus*. Cambridge University Press, 2007.

[Mil92] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-calclulus*. Cambridge University Press, 1999.

[Nes00] Uwe Nestmann. What is a "good" encoding of guarded choice? *Inf. Comput.*, 156(1-2):287–319, 2000.

[Pal03] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003.

[PH01] C. Palamidessi and O. Herescu. A randomized encoding of the -calculus with mixed choice, 2001.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus:A Theory of Mobile Processes*. Cambridge University Press, 2001.

[Tur96] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, 1996.