$\pi_{\texttt{dist}}$: Towards a Typed π-calculus for Distributed Programming Languages

MSc Thesis *(Afstudeerscriptie)*

written by

Ignas Vyšniauskas
(born May 27th, 1989 in Vilnius, Lithuania)

under the supervision of **Dr Wouter Swierstra** and **Dr Benno van den Berg**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
Jan 30, 2015	Dr Benno van den Berg
	Prof Jan van Eijck
	Prof Christian Schaffner
	Dr Tijs van der Storm
	Dr Wouter Swierstra
	Prof Ronald de Wolf

Institute for Logic, Language and Computation

Abstract

It is becoming increasingly clear that computing systems should not be viewed as isolated machines performing sequential steps, but instead as cooperating collections of such machines. The work of Milner and others shows that 'classical' models of computation (such as the λ-calculus) are encompassed by suitable *distributed* models of computation. However, while by now (sequential) computability is quite a rigid mathematical notion with many fruitful interpretations, an equivalent formal treatment of distributed computation, which would be universally accepted as being canonical, seems to be missing.

The goal of this thesis is not to resolve this problem, but rather to revisit the design choices of formal systems modelling distributed systems in attempt to evaluate their suitability for providing a formals basis for distributed programming languages. Our intention is to have a minimal process calculus which would be amenable to static analysis. More precisely, we wish to harmonize the assumptions of π-calculus with a linear typing discipline for process calculi called *Session Types*.

We begin by discussing and comparing various process calculi, both in purely theoretical and in pragmatic terms. In doing so, we discover an interesting misalignment between some folklore interpretations of results in the literature on process calculi, which stems from a lack of sufficient criteria for relating process languages.

The comparison leads us to a restricted subset of π-calculus which we call $\pi_{\texttt{dist}}$, reminiscent of Merro and Sangiorgi's Lπ-calculus [MS04] in terms of semantics and of Fournet and Gonthier's Join-calculus [FG00] in spirit. We describe the reduction and transition semantics of this calculus, develop some of its theory and discuss its relative expressiveness. In particular, we show that $\pi_{\texttt{dist}}$ is more *distributable* than the asynchronous π-calculus.

We then describe a minimal programming language, TinyPi, in order to abstract away from the syntax of π-calculus and more faithfully expresses the intended semantics of $\pi_{\texttt{dist}}$. With the aim of providing formal guarantees about $\pi_{\texttt{dist}}$ programs, we investigate the application of Session Types. In particular, we show that the type system πDILL of Caires and Pfenning's [CP10] – with some small modifications – can be used to type process communication under asynchronous FIFO semantics, while preserving full type safety.

Acknowledgements

The road to this thesis has not been a short one and to give full acknowledgements is entirely impossible, however a few names deserve particular mentions.

I would like to first and foremost thank my supervisor, Wouter Swierstra. Firstly, for his calmness and faith in me throughout my thesis writing. However, much more importantly, I would like to thank Wouter for being my academic guide in the last 2 years by continuously inspiring and selflessly providing me with opportunities to explore and learn about topics I would have otherwise had no chance to.

Additionally, I would like to thank my thesis committee for their hard work and willingness to examine my thesis. I am also very thankful to my academic mentor Benedikt Löwe for his patience, acumen and excellent guidance throughout the years. Many thanks also go to Tanja Kassenaar for her optimism and generous helpfulness.

Finally, I would like to thank Johannes Emerich and Yuning Feng not only for their close friendship and empathy, but especially for their sincere attempts to understand my incoherent ramblings and discuss my thesis-related problems, which helped me dismiss countless bad ideas. These acknowledgements would also not be complete without a truly heartfelt thank you to my partner, Rūta, for her unconditional support and trust in me.

I am also deeply grateful to the rest of the staff and the whole community of the ILLC who have managed to form an unmatchable environment where multidisciplinary, multicultural and classless collaboration flourishes resulting in beautiful friendships and excellent academic work.

Contents

1 Introduction and background

1.1 Motivation

We will not keep it a secret that, at least in some respects, this thesis is an attempt to synchronise aspects of process calculi and type theory research with their applications. In particular, we hope that this work can serve as an inspiration to reconsider some traditional approaches in implementing and verifying programming languages designed for distributed communication and concurrency.

It is at least slightly surprising that more than 20 years after the introduction of π-calculus by Milner, Parrow, and Walker [MPW92] (and almost 40 years after the introduction of its predecessor CCS [Mil80]), which has seen extensive developments of its theory and had quite a few experimental implementations, there are almost no languages adopted by the industry based *primarily* on some process calculus[1]. Although this is not a very long timespan, it is still in contrast with the adoption of other parts of Milner's work, such as his work on the ML programming language, which spawned entire families of well-adopted programming languages.

Why is this so? On a very high level it can be argued that problems related to concurrency and distribution were not as ubiquitous before the appearance of the internet, many-core processors and mobile devices. This means that both the motivation and intuition for concurrency-first based design of programming languages was missing.

However, another possible issue is that the majority of the process calculi build upon synchronous communication primitives[2]. For instance, Milner's basic idea was that "communication is interaction" and he often used a phone conversation as typical example of synchronous communication. It is unlikely that this idea would have appealed to someone who has ever participated in an online voice-chat over a slow internet connection – the illusion of synchronous or "live" communication is shattered as soon as there is any jitter. As we will argue throughout this thesis, this implies a rather unnatural interpretation of communication and, worse, makes it hard to provide a concrete implementation for such primitives.

It is thus our duty to at least attempt to fix this. Therefore, in the first part of the thesis we try to embrace asynchrony and examine what implications asynchrony has to various aspects of π-calculus. Most of our observations are not novel in any way, however we attempt to give a fresh view of the features of

[1] There are two exceptions which come very close though. The first is the Erlang programming language [Arm07] which, although not modelled after, comes very close to the Actor model and has processes as the minimal building blocks. The second is the Go programming language [GPT07], which, while not being process-centric, implements the π-calculus with mixed choice. However, the first is not based on any formal process calculus, while the second merely contains extensive features of π-calculus, but is not designed from the ground-up as an implementation of it. In particular, it is based on a shared memory model. Both situations imply that static analysis is very complicated for these languages.

[2] In fact, a purely asynchronous variant of π-calculus was only formulated in the 90's [Bou92][HT91].

π-calculus in terms of their feasibility for a fully distributed implementation. As we later argue, this at least partially coincides with the degree of *distributability* of a process calculus.

A similar issue due to the synchronous foundations re-appears when we investigate options for achieving provable safety guarantees of distributed programs (in languages based on process calculi). A state of the art technique for achieving this is equipping process calculi with so-called Session Type systems. In here we discover an interesting phenomena: although Session Type systems were initially applied to synchronous calculi, there appears to be a trend in shifting to calculi where communication is asynchronous, but preserves message order. It is a tempting challenge to align and exploit this observation by identifying a suitable flavour of π-calculus and endowing it with an appropriate kind of Session Type system, which is what we do in the second part of the thesis.

1.2 Basic notions of concurrency and distribution

Before we proceed with any technical considerations, it is good to fix terminology to avoid unnecessary confusion about the subject matter. For this purpose, we recall some standard notions from distributed systems theory.

1.2.1 Asynchrony and causality

Informally, *synchrony* vs. *asynchrony* refers to the distinction of two actions happening instantaneously and with some (finite) delay. In practice, this usually means "within a negligible time-frame" and "with an observable delay" respectively.

In the setting of concurrency theory and communicating systems we are usually concerned with the asynchrony (or synchrony) of *message delivery*. In fact, one can describe the *degree* of (a)synchrony of a message passing system, by considering the possible observations (regarding message input or output) that can be made. For the sake of providing a more formal description, we assume a system of named processes which communicate via message-passing over some abstract communication medium. Moreover, a partial ordering ("global time") of events is assumed and for simplicity we assume that in the following text all input actions, or *receives*, happen within a single process. Usually, 4 degrees of asynchrony are identified, which form the following (strict) hierarchy[3]:

Asynchronous A system is said to be (fully) asynchronous if there are no constraints on message delivery (except that message input and output should respect the arrow of time). Most importantly, the event of message reception can happen at any later point in time than the output event.

FIFO-order FIFO stands for First-In First-Out and is used to denote the fact that, within a channel, messages *between two processes* are received in the same order they were output. It is implicitly assumed that there is some

[3]This hierarchy and its properties are excellently described in the paper "Synchronous, Asynchronous, and Causally Ordered Communication" [CMT96] to which we refer the reader to for examples and additional details.

kind of communication *buffer*, or more specifically a *queue*, in-between the processes, which preserves the message order.

Causally-ordered Communication is said to be causally-ordered when the order of message inputs (within a single process) always respects the order of outputs (of potentially more than one process) with respect to the global time. This can be understood as a strengthening of FIFO-ordering, where we generalise the preservation of the order of outputs of a single process to preservation of outputs among *all* processes. In practice this means that even if two completely unrelated processes send messages to the same process, the order of inputs must preserve the ordering of these output actions according to the global time.

Synchronous From [Bou87]:

> A system has synchronous communications if no message [...] can be sent along a channel before the receiver is ready to receive [...] on the channel. For an external observer, the transmission then looks instantaneous and atomic. Sending and receiving a message correspond in fact to *the same event*.

(Emphasis added.)

In other words, communication can be said to be synchronous if no other action can happen (strictly) in-between the input and output of a message.

When described in this way, it is clear that asynchronous communication is the least constrained type and therefore represents the largest class of communication systems, while synchronous communication is on the opposite end of the spectrum. On the other hand, synchronous communication provides the strongest *guarantees* about message delivery and is therefore intuitively the most powerful framework (in terms of absolute expressiveness). In fact, as we will later discuss, synchronous *process calculi* with certain features are strictly separated from asynchronous calculi. In particular, this means that certain communication protocols which can be expressed in synchronous calculi, cannot be expressed in asynchronous ones.

The natural question here, which we will examine in further sections more concretely in the context of π-calculus, is: when formally modelling distributed systems, what degree of asynchrony should be assumed? Clearly, very often the answer depends on the problem domain, however it is is still a pressing question if we are interested in some foundational theory of communication.

Process algebraists tend to choose one of the two extremes: fully synchronous or fully asynchronous communication, with a visible preference for the former option. Undoubtedly, these are sane options. However, in distributed systems research, a preference for the latter can be observed, mostly because it reflects usual constraints occurring in applications. Nevertheless, it can also be observed that, if stronger delivery guarantees are needed, quite often FIFO delivery is assumed.

The author of this thesis believes that FIFO-ordered communication is a very 'optimal' assumption in terms of balance between the added strength to the

formal framework and its applicability. Philosophical arguments aside, FIFO-ordered communication can be identified as an *upper-bound* on the guarantees which can be provided in a distributed setting, without requiring global coordination.

Intuitively, fully asynchronous message delivery can be interpreted as "communication via bags": the messages output by one process are put in a bag and later retrieved from the bag by another process. The order in which the messages are fetched from the bag is therefore completely undetermined and there is no need for some global entity to maintain some invariants about how multiple processes interact with the bag[4]. This interpretation is well-known to process algebraists and can be formally shown to be precise [BPV08].

Similarly, FIFO-ordered communication can be interpreted as communication via queues or via belts: a sent message is queued up and can only be retrieved after all the messages in front of it are processed. As long as messages from one queue are retrieved by a *single* process, there is also no need for processes to coordinate when communicating. Note that, once again, the queue does not have to exist as some centralised component in the system (which would defeat the claim that this type of communication does not require coordination): it is sufficient that, for example, the sender tags the messages with: 1) his own (unique) name and; 2) a sequence number. In that case, the receiver can simply buffer the messages locally so that they are processed in their output sequence order.

Note that there is no such mechanism for causally-ordered delivery: since processes execute at independent paces and by assumption share no state (including separate processor clocks), given two messages sent by two different processes, there is no way to determine which of them sent the message first without either: a) them communicating with each other to resolve the ambiguity or; b) sending itself relying on some coordinator that determines which of the processes is the first to send the message.

Given these observations, it is then not surprising at all that the most popular network transport protocol – TCP [Ste93] – provides exactly FIFO-ordered delivery guarantees. Moreover, the universal presence of TCP in computer networking settings implies that theoretical work assuming FIFO delivery is very easy to implement in a wide range of practical environments.

Curiously, FIFO-ordered delivery has been examined very little in the setting of process calculi, mostly because it has been argued that FIFO communication can be simulated via intermediate queue processes. However, in many ways this is an overly simplistic view that fails to harvest the benefits of such assumptions. In particular, later we will show that "naively" adding FIFO-ordering to π-calculus results in quite an unpleasant formal system. We therefore think that an in-depth analysis of the theory of process calculi communicating using explicitly FIFO-ordered mediums could be very fruitful.

[4]If the word 'bag' implies central coordination to you, you can replace it with the word 'ether'.

1.3 Process calculi

We now take a look at the formal approaches to modelling communication and distributed process systems. Many formalisms have been proposed for this task: Petri nets [Pet62], Kahn process networks [Kah74], trace theory [Maz86] and communicating finite-state machines [BZ83] to name a few of the most well-known ones. All of these formalisms have been studied extensively and all of them were successfully applied in solving real-world problems.

There is one more branch which stands out for its enormous success in computational contexts and applications – *process calculi* or *process algebras*. Process calculi form a diverse family of formalisms, which share in common an algebraic formulation that permits equational reasoning. Moreover, most of the process calculi aim to describe fully concurrent models of computation and therefore operate by means of message passing on *names* or *channels*, which are the only shared 'component' between processes.

The most well known process calculi are Hoare's Communicating Sequential Processes (CSP, [Hoa78]), Bergstra and Klop's Algebra of Communicating Processes (ACP, [BK84]) and Milner's Calculus of Communicating Systems (CCS, [Mil80]), which was later developed into the π-calculus [Mil99]. The developments of these calculi were greatly influenced by each other, however they did not converge into some unified framework. In fact, attempts to precisely describe the relative expressiveness of these calculi continue up to this day and different process algebraists have different opinions ([Par14], [Gor10], [GGS13], [Gla12], [Pal03], [Par08], [Pet12]) about it.

The word *calculus* in both CCS and π-calculus reflects Milner's foundational aspirations while developing these theories:

> *This is why we call it a "calculus". We dare to use this word by analogy with Leibniz's differential calculus; the latter — incomparably greater — is based upon continuous mathematics, while the π-calculus is based upon algebra and logic; but the goal in each case is analysis, in one case of physical systems and in the other case of informatic systems.*
>
> *Moreover, we found that the π-calculus may be significant in a way which we did not fully expect. We are very familiar with basic models of computation; for example, most scientists have heard of Alan Turing's famous "Turing Machine", a very simple device on which anything that's computable can be computed. Is there such a basic model for all discrete interactive behaviour? We don't yet know how to pose this question precisely, but we think of the π-calculus as a tentative step towards such a model.*

(From *Speech by Robin Milner on receiving an Honorary Degree from the University of Bologna* [Mil97]).

Milner was inspired[5] by the Actor model [Agh86] to come up with a formal

[5] From his Turing Award Lecture [Mil93, p. 86]:

> Now, the pure λ-calculus is built with just two kinds of thing: terms and variables. Can we achieve the same economy for a process calculus? Carl Hewitt,

language which would be to concurrent computation what λ-calculus is to sequential computation. This culminated in the formulation of the π-calculus, where all the syntactic levels of CCS were collapsed into a single notion of a *name*. Names therefore denote the communication locations (or *channels*), form the objects of the content being communicated and determine the control flow of processes, which allows for a very uniform treatment of processes in the language of π-calculus.

Most importantly, the fact that distinguishes π-calculus from other process calculi (and, in fact, most of the previously mentioned formalisms) is the ability to directly express *mobility* by allowing to pass around and communicate over the transmitted names. This is a crucial feature of π-calculus which gives it enormous expressive power and which is essential for modelling distributed computing systems with varying topologies and varying local *knowledge*. More over, name passing, when combined with name *restriction*, is also a convenient means of achieving *process abstraction*.

The author of this thesis shares Milner's appreciation of the λ-calculus and interests in foundational aspects of communication. Moreover, the mobility feature of π-calculus is essential for describing many fundamental concepts of distributed systems theory, such as problems of consensus and leader-election. Finally, π-calculus has quite recently re-appeared in the setting of programming language theory, as the target of Session Typing systems. Due to these reasons, π-calculus is a natural fit for our needs and will be the language of our choice for the rest of this thesis.

1.4 Quick introduction to the π-calculus

In this section we briefly review the main concepts of π-calculus.

Similarly to how the essence of λ-calculus can be said to reside within substitution, which manifests itself via the β-reduction rule

$$(\lambda x.M)N \xrightarrow{\beta} M\{N/x\}$$

at the heart of the π-calculus is the interaction or synchronisation reduction rule

$$\overline{x}\langle y\rangle.P \mid x(z).Q \longrightarrow P \mid Q\{y/z\}$$

which exhibits a form of substitution, except that now variables are replaced only by *names*, instead of arbitrary terms.

This rule is understood to mean that two processes running in parallel ('|' is the parallel composition operator) interact, or *synchronise*, if one of them offers an output capability (denoted by '$\overline{x}\langle y\rangle$'), while the other offers an input capability (denoted by '$x(z)$') on the same *name* x. Given such a match of capabilities, the name y can be communicated from one process to the other, by replacing

with his Actors model, responded to this challenge long ago; he declared that a value, an operator on values, and a process should all be the same kind of thing: an *actor*. This goal impressed me, [..]

Figure 1 Syntax of π-calculus

Names

$$\mathcal{N} = \{a, b, \ldots, x, y, \ldots\}$$

Prefixes

$$\begin{array}{rcll} \pi, \alpha, \beta & ::= & \overline{x}\langle y\rangle & \text{output} \\ & | & x(y) & \text{input} \\ & | & \tau & \text{internal action} \end{array}$$

Process syntax

$$\begin{array}{rcll} P, Q, R & ::= & (\nu x)P & \text{name restriction} \\ & | & \pi.P & \text{prefixing} \\ & | & P + Q & \text{choice} \\ & | & P \mid Q & \text{parallel composition} \\ & | & \mathsf{end} & \text{empty process} \\ & | & !P & \text{replication} \end{array}$$

the bound occurrences (of the name z) in the receiving process. The operator '.' denotes prefixing or sequencing.

The intuition behind the interaction rule is that in order to communicate, two parallel processes have to first synchronise, at which stage an instantaneous information exchange is performed. Moreover, since this is the only rule which performs substitution, it is 'responsible' for all of the computational content of π-calculus as well. To paraphrase Milner: interaction is computation, and *vice versa*.

Another crucial concept of the π-calculus is *name* or *scope restriction*, represented by the operator ν. The notation $(\nu y)P$ implies that the name y is a *fresh* (or a *new*) name in the process P. Read differently, the free name y in P becomes *restricted* in $(\nu y)P$. The freshness or restriction implies that the name y is no longer available for any interaction, that is, for any composition $Q \mid (\nu y)P$, no communication can happen via the name y. This is also called *hiding* in π-calculus and is used to achieve a form of abstraction by hiding communication internal to some process. As an example, in

$$P \mid (\nu x)(\overline{x}\langle y\rangle.Q_1 \mid x(y).Q_2 \mid c(z).Q_3)$$

the communication over x is internal to the restricted scope, however P might still be able to communicate over the channel c.

We briefly recall the syntax and reduction semantics of the monadic π-calculus with choice. Our presentation is standard and mostly based on the one in [SW01], but to keep the thesis self-contained we explicitly state the definitions we will use throughout it. Figure 1 presents the syntax of π-calculus. We let $\mathcal{N}$ denote an infinite set of *names*, which we will also call *channels* interchangeably. The operators $(\nu y)P$ and $x(y).P$ are both binding occurrences of y in P. Throughout the thesis we assume Barendregt's variable convention [Bar85, Definition 2.1.13]: bound variables are assumed to always be distinct from free variables and binders never bind already bound variables in scope. We

Figure 2 Structural rules of π-calculus

$$P \mid Q \equiv Q \mid P \qquad P + Q \equiv Q + P$$
$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad P + (Q + R) \equiv (P + Q) + R$$
$$\mathsf{end} \mid P \equiv P \qquad \mathsf{end} + P \equiv P$$
$$(\nu c)(P \mid Q) \equiv (\nu c)P \mid Q,\ c \notin \mathsf{fn}(Q)$$
$$(\nu c)(P + Q) \equiv (\nu c)P + Q,\ c \notin \mathsf{fn}(Q) \qquad !P \equiv P \mid !P$$
$$(\nu c)(\nu c')P \equiv (\nu c')(\nu c)P$$
$$(\nu c)\mathsf{end} \equiv \mathsf{end}$$

Figure 3 Reduction semantics of π-calculus

$(\overline{c}\langle d\rangle.P + P') \mid (c(x).Q + Q')$	$\longrightarrow$	$P \mid Q\{d/x\}$	[Comm]
$\tau.P + Q$	$\longrightarrow$	P	[Tau]
$P \longrightarrow P'$	implies	$(\nu c)P \longrightarrow (\nu c)P'$	[Scop]
$P \longrightarrow P'$	implies	$P \mid Q \longrightarrow P' \mid Q$	[Par]
$P \equiv P', Q \equiv Q'$ and $P' \longrightarrow Q'$	implies	$P \longrightarrow Q$	[Str]

often omit the termination continuation end and assume the following operator precedence in ascending order: '|', 'ν', '+', '!' and '.'. In particular, $(\nu x)!\pi.P + Q \mid R = (\nu x)(!(\pi.P) + Q) \mid R$.

π-calculus processes are considered modulo renaming (α-conversion) and the (smallest) congruence relation generated by the structural rules listed in Figure 2. The structural rules guarantee that $\equiv$ is closed under commutative monoid laws for both parallel composition ($|$) and sum ($+$), with end as unit. Figure 3 presents the *reduction semantics* of π-calculus.

A useful fact is the following:

Definition 1.4.1 (Standard form [Mil99, Def.. 9.12])**.** A process P is said to be in *standard form* if

$$P = (\nu\vec{n})(P_1 \mid P_2 \mid \cdots \mid P_n \mid !R_1 \mid \ \cdots \mid !R_m)$$

where each P_i is a non-empty sum and all R_k's are in standard form already.

Proposition 1.4.1 (Existence of standard form [Mil99, Prop. 9.13])**.** *For every process P, there exists a process P' in standard form, such that $P \equiv P'$.*

The reduction relation $\longrightarrow$ is defined as the smallest relation on process terms generated from the rules in Figure 3. Throughout the thesis we will use juxtaposition of relations to denote their composition, for example $P \longrightarrow \equiv P'$ means that there exists a process Q, such that $P \longrightarrow Q$ and $Q \equiv P'$. Moreover, we will use the notation $P\mathcal{R}$ to denote that there exists some P' such that $P\mathcal{R}P'$. For example, $P \longrightarrow$ means that P *reduces*.

Reduction rules only describe the transitions of closed terms. In order to consider the transitions of open terms in arbitrary contexts, π-calculus is equipped

with a labelled transition system (Figure 4). We re-use the prefix names for transition labels, which are described by the following grammar:

$$\begin{array}{lcll} \alpha & ::= & \overline{x}\langle y\rangle \mid x(y) & \text{– free input/output} \\ \alpha' & ::= & (\nu y)\overline{x}\langle y\rangle & \text{– bound output} \\ & \mid & \alpha & \\ \mu & ::= & \alpha' \mid \tau & \text{– action} \end{array}$$

Moreover, $\underline{\alpha}$ and $\overline{\alpha}$ are assumed to denote arbitrary input and output labels respectively. To avoid any confusion, following [SW01], we define functions related to names of actions and prefixes:

Definition 1.4.2 (Name terminology)**.**

$$\begin{array}{rcll} \mathsf{subj}(\overline{x}\langle y\rangle) & = & \mathsf{subj}(x(y)) = x & \text{– subject} \\ \mathsf{subj}((\nu x)\alpha) & = & \mathsf{subj}(\alpha) & \\ \\ \mathsf{obj}(\overline{x}\langle y\rangle) & = & \mathsf{obj}(x(y)) = y & \text{– object} \\ \mathsf{obj}((\nu x)\alpha) & = & \mathsf{obj}(\alpha) & \\ \\ \mathsf{fn}(\tau) & = & \emptyset & \text{– free names} \\ \mathsf{fn}(\alpha) & = & \{\mathsf{subj}(\alpha), \mathsf{obj}(\alpha)\} & \\ \mathsf{fn}((\nu x)\alpha) & = & \mathsf{fn}(\alpha) \setminus \{x\} & \\ \\ \mathsf{bn}(\tau) & = & \mathsf{bn}(\alpha) = \emptyset & \text{– bound names} \\ \mathsf{bn}((\nu x)\alpha) & = & \mathsf{fn}(\alpha) \cap \{x\} & \\ \\ \mathsf{names}(\mu) & = & \mathsf{bn}(\mu) \cup \mathsf{fn}(\mu) & \text{– names} \end{array}$$

We use the so-called *early* formulation of the LTS. Since we consider processes modulo structural congruence, we include the rule Cong. This allows us to consider derivations modulo structural congruence and avoid duplicate symmetric left-right rules for Sum, Par, Open and Close.

We recall an important fact about the LTS from [SW01], which says that the reduction relation coincides with the τ-transitions of the LTS:

Lemma 1.4.1 (Harmony lemma [SW01, Lemma 1.4.15])**.**

1. $Q \equiv P \xrightarrow{\mu} P'$ *implies* $Q \xrightarrow{\mu} Q' \equiv P'$
2. $P \longrightarrow P'$ *iff* $P \xrightarrow{\tau} Q \equiv P'$

The main reason for introducing an LTS is because it permits reasoning about the behaviour of *open* process terms in different contexts. In particular, it enables the definitions of behavioural process equivalences that allow to compare and relate processes in terms of their observable behaviour.

Below we represent a few standard equivalences and related notions which will be used throughout this thesis. We refer the reader to [SW01][ACS96][FG05] for extended definitions and in-depth discussions.

Definition 1.4.3 (One-hole context)**.** A process term C with a single occurrence of end replaced by a *hole*, (denoted by $[\cdot]$) is called a *context* and is denoted

Figure 4 LTS of π-calculus

$$\frac{}{\overline{x}\langle y\rangle.P \xrightarrow{\overline{x}\langle y\rangle} P}\ \text{(Out)} \qquad \frac{}{x(z).P \xrightarrow{x(y)} P\{y/z\}}\ \text{(Inp)}$$

$$\frac{}{\tau.P \xrightarrow{\tau} P}\ \text{(Tau)} \qquad \frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P' + Q}\ \text{(Sum)}$$

$$\frac{P \xrightarrow{\mu} P' \qquad \mathsf{bn}(\mu) \cap \mathsf{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\mu} P' \mid Q}\ \text{(Par)}$$

$$\frac{P \xrightarrow{\overline{x}\langle y\rangle} P' \qquad Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}\ \text{(Comm)} \qquad \frac{P \xrightarrow{\overline{x}\langle y\rangle} P' \qquad y \neq x}{(\nu y)P \xrightarrow{(\nu y)\overline{x}\langle y\rangle} P'}\ \text{(Open)}$$

$$\frac{P \xrightarrow{(\nu z)\overline{x}\langle z\rangle} P' \qquad Q \xrightarrow{x(z)} Q' \qquad z \notin \mathsf{fn}(Q)}{P \mid Q \xrightarrow{\tau} \nu z(P' \mid Q')}\ \text{(Close)}$$

$$\frac{P \xrightarrow{\mu} P' \qquad z \notin \mathsf{names}(\mu)}{(\nu z)P \xrightarrow{\mu} (\nu z)P'}\ \text{(Res)}$$

$$\frac{P \equiv P' \qquad P \xrightarrow{\mu} Q \qquad Q \equiv Q'}{P' \xrightarrow{\mu} Q'}\ \text{(Cong)}$$

by $C[\cdot]$. Contexts can be applied (notation $C[A]$) to other contexts and process terms, which results in the hole being replaced by the subject and produces new contexts or terms.

For example, from $(\nu y)(\overline{x}\langle y\rangle.\mathsf{end} \mid \mathsf{end})$ we can obtain the contexts $C[\cdot] = (\nu y)(\overline{x}\langle y\rangle.[\cdot] \mid \mathsf{end})$ and $C'[\cdot] = (\nu y)(\overline{x}\langle y\rangle.\mathsf{end} \mid [\cdot])$. They can then be composed via application, e.g. $C[C'[P]] = (\nu y)(\overline{x}\langle y\rangle.(\nu y)(\overline{x}\langle y\rangle.\mathsf{end} \mid P) \mid \mathsf{end})$.

Definition 1.4.4 (Strong bisimilarity)**.** Strong bisimilarity is the largest symmetric relation, $\sim$, such that whenever $P \sim Q$, then $P \xrightarrow{\mu} P'$ implies $Q \xrightarrow{\mu} Q'$ and $P' \sim Q'$.

Definition 1.4.5 (Strong barbed bisimilarity)**.** Strong barbed bisimilarity is the largest symmetric relation, $\overset{\bullet}{\sim}$, such that whenever $P \overset{\bullet}{\sim} Q$, then

1. If $P \xrightarrow{\alpha_1}$ then $Q \xrightarrow{\alpha_2}$ where $\mathsf{subj}(\alpha_1) = \mathsf{subj}(\alpha_2)$.
2. $P \longrightarrow P'$ implies $Q \longrightarrow Q'$ and $P' \overset{\bullet}{\sim} Q'$.

Definition 1.4.6 (Strong barbed equivalences)**.** Two processes P, Q are *strong barbed equivalent*, $P \simeq Q$, if $P \mid R \overset{\bullet}{\sim} Q \mid R$ for all R.

Definition 1.4.7 (Strong barbed congruence)**.** Two processes P, Q are *strong barbed congruent*, $P \simeq^c Q$, if for all contexts C, $C[P] \overset{\bullet}{\sim} C[Q]$.

The term *strong* in the above definitions refers to the fact that they are based on one-step transitions. That is, if we consider the above definitions as bisimulation games, then the defender has to reach a matching state via a single transition.

However, as reductions or τ-transitions are considered to be internal or hidden, it makes sense to ignore them and allow the defender to perform extra internal reductions. This leads to the notions of weak transitions and weak bisimilarities.

Definition 1.4.8 (Weak reduction and weak transition)**.** Define the

Weak reduction relation $\Longrightarrow$ as the reflexive transitive closure of the reduction relation $\longrightarrow$, i.e.

$$\Longrightarrow \;=\; \longrightarrow^*$$

Weak transition relation $\overset{\mu}{\Longrightarrow}$ as

$$\overset{\mu}{\Longrightarrow} \;=\; \Longrightarrow\xrightarrow{\mu}\Longrightarrow$$

Using these relations, we can define the respective weak versions of the equivalences above, denoted by $\approx$, $\overset{\bullet}{\approx}$ and $\cong$ and $\cong^c$, where in the bisimulation game the defender can respond using weak transitions rather than strong ones. We only present weak bisimilarity as an example:

Definition 1.4.9 (Weak bisimilarity)**.** Weak bisimilarity is the largest symmetric relation, $\approx$, such that whenever $P \approx Q$, then $P \xrightarrow{\mu} P'$ implies $Q \overset{\mu}{\Longrightarrow} Q'$ and $P' \approx Q'$.

For example, the processes $P = \overline{x}\langle y\rangle$ and $Q = (\nu z)(\overline{z}\langle o\rangle \mid z(u).\overline{x}\langle y\rangle)$ are weakly barbed congruent, but not even strongly bisimilar.

1.5 The many faces of π-calculus

We now attempt to dissect π-calculus into its core building blocks in order to, first of all, gain a deeper understanding of their significance, but also to follow our wish to identify a minimal sub-calculus with sufficient expressive power. Most of the observations in this section are not novel in any way and only summarise a vast collection of different approaches at comparing variants of the π-calculus.

As already hinted at previously, a fascinating fact about the π-calculus is the power of its name restriction operator. In particular, it permits a very powerful form of abstraction, which allows to encode or simulate many of the stronger π-calculus primitives inside its weaker sub-calculi. To continue our parallel with λ-calculus, one can compare this situation with, for example, including boolean conditionals and natural numbers as primitive constructs versus simulating them via their Church-encodings.

The variant of π-calculus described in the previous section is known in the literature as *π-calculus with (free) choice*, which we will denote as π_{fc}. Below we present a few of the most well known sub-calculi of π_{fc}, together with some extensions. We now give their brief descriptions and later discuss the relations between them in the next section.

The (guarded) mixed-choice π_{mc}-calculus

The (guarded) mixed-choice calculus is obtained from π_{fc} by requiring all summands in process sums to be *guarded* (i.e. prefixed) by an input or output action. This introduces a single syntactic restriction that $P = P_1 + P_2$ implies $P \equiv \alpha.P + \beta.Q$, where α, β are either input or output prefixes. In particular, τ prefixing and unprefixed summands are disallowed. The word *mixed* refers to the fact that both input and output prefixes can appear as part of the same sum.

Guardedness is an important notion in process algebras and becomes particularly important when considering recursive processes. In short, making every summand guarded ensures that branch choices are always observable. In many settings, this greatly simplifies provability of termination and progress by reducing these issues to productivity of the terms.

The separate-choice π_{sc}-calculus

The separate-choice calculus is a restriction of π_{mc} where all prefixes in a sum must be of the same kind, i.e. every sum consists of either only input-guarded or only output-guarded summands.

The input-guarded choice π_{ic}-calculus

π_{ic} is a sub-calculus of π_{sc} in which only input-guarded sums are permitted.

The choice-free π_{nc}-calculus

π_{nc} is the sub-calculus of $\pi_i c$ which contains no sum operation at all.

The asynchronous π_a-calculus

The asynchronous π_a calculus deserves a slightly more elaborate description, because although it is a very straightforwardly defined sub-calculus of π_{nc}, its overall flavour and theory is very different from the synchronous variants of π-calculus.

Viewed as a message-passing system, the π-calculus is a model of *synchronous* communication. This can be seen from the reduction rules, in particular the rule [Comm] implies that sending and receiving is an instantaneous operation, which means message delivery is synchronous according to the classification in subsubsection 1.2.1.

One way to simulate asynchrony in π-calculus is to perform communication between processes via intermediate buffers. In the case where a channel is only shared between two processes, one can split the channel into two input and output channels and represent the (asynchronous) buffer as a replicated process which forwards messages from the output part to the input part. Then the usual instantaneous communication happens in multiple steps:

$$\begin{aligned} \overline{x_o}\langle y\rangle.P \mid x_i(z).Q \mid !x_o(z).\overline{x_i}\langle z\rangle &\longrightarrow P \mid x_i(z).Q \mid !x_o(z).\overline{x_i}\langle z\rangle \mid \overline{x_i}\langle y\rangle \\ &\longrightarrow P \mid Q\{y/z\} \mid !x_o(z).\overline{x_i}\langle z\rangle \end{aligned}$$

Note that after the first reduction, P could, for example, output another message y' on the channel x_o and it could be received by Q before the name y, because interaction with any of them would be possible. This shows that such an encoding simulates asynchronous message delivery according to the specification in subsubsection 1.2.1.

This example provides a hint on how to obtain a sub-calculus of π_{nc}, such that *all* communication is asynchronous. In fact, buffers are not even needed: due to the uniform nature of π-calculus, messages-in-transit can be represented as "atomic" processes which need to be interacted with themselves, such as $\overline{x_i}\langle y\rangle$ in the above example. It was first observed by Honda and Tokoro that it is sufficient to forbid output prefixing in order to obtain a fully asynchronous subcalculus of π_{nc} [HT91]. Namely, in π_a instead of output prefixing, we only have output actions $\overline{x}\langle y\rangle$.

Surprisingly, this simple restriction is sufficient to model asynchronous communication, without making any further changes to the reduction or transition semantics. However, the behavioural flavour of π_a is very different from the synchronous variants. First of all, output guarded sums no longer exist because output prefixing is not available and, in fact, only input-guarded sums are considered justifiable in π_a. In fact, π_a is normally presented as a sub-calculus of π_{nc}, i.e. without any sums at all. We postpone the discussion of sums in the context of π_a until later.

One source of differences in the theory of π_a is that the processes equivalences used for synchronous π-calculus calculi are not suitable for π_a. Instead, a "natural" process equivalence is considered to be induced by the following bisimilarity.

Definition 1.5.1 (Weak asynchronous bisimilarity)**.** Weak asynchronous bisimilarity is the largest symmetric relation, $\approx_a$, such that whenever $P \approx_a Q$, then $P \xrightarrow{\mu} P'$ implies either:

1. $Q \stackrel{\mu}{\Longrightarrow} Q'$ and $P' \approx_a Q'$
2. If $\mu = x(y)$, then $Q \Longrightarrow Q'$ and $P' \approx_a Q' \mid \overline{x}\langle y \rangle$

and the following barbed version:

Definition 1.5.2 (Weak asynchronous barbed bisimilarity)**.** Weak asynchronous barbed bisimilarity is the largest symmetric relation, $\stackrel{\bullet}{\approx}_a$, such that whenever $P \stackrel{\bullet}{\approx}_a Q$, then

1. If $P \xrightarrow{\overline{\alpha_1}}$ then $Q \stackrel{\overline{\alpha_2}}{\Longrightarrow}$, such that $\mathsf{subj}(\alpha_1) = \mathsf{subj}(\alpha_2)$.
2. $P \longrightarrow P'$ implies $Q \Longrightarrow Q'$ and $P' \stackrel{\bullet}{\approx}_a Q'$.

Analogously, asynchronous variants of barbed equivalence and barbed congruence are defined.

These behavioural equivalences capture the intuition that, in asynchronous contexts, it is not possible to observe inputs because they happen independently from message output. In particular, the following is a characterising equivalence of π_a:

$$\varepsilon(a) := a(x).\overline{a}\langle x \rangle \cong_a^c \mathsf{end} \tag{1.5.1}$$

An important consequence of this equivalence is that it permits to introduce the 'retransmission' processes $\varepsilon(a)$ into any context without affecting the observable behaviour or process terms, i.e.

$$\varepsilon(a) \mid P \cong_a^c \mathsf{end} \mid P \equiv P$$

This leads to powerful process constructions, such as as an "equator" process (due to Honda and Yoshida [HY95]), defined as:

$$\mathrm{Eq}(a, b) := !a(x).\overline{b}\langle x \rangle \mid !b(x).\overline{a}\langle x \rangle$$

Equator processes can be introduced into arbitrary contexts to simulate the effect of name substitution. It can be shown (see [Mer99] for details) that the following equivalence holds:

$$\mathrm{Eq}(a, b) \mid P \cong_a^c P\{b/a\}$$

The localised Lπ-calculus

The localised Lπ-calculus is usually considered as a sub-calculus of π_a, though its restriction is orthogonal to the other ones. In Lπ, only the output capability of names is allowed to be transmitted. This can be realised either with the help of a type system, or by syntactically disallowing x to appear as an input subject in the continuation P of the process $a(x).P$. The idea is that names represent channels which have a location and therefore can only be *read* by the process which controls it. These kind of situations often arise in real distributed systems, which is why Lπ is mostly considered as an asynchronous sub-calculus.

The internal mobility Iπ-calculus

Similarly to the local channel restriction of Lπ, the internal mobility calculus Iπ permits only the output of *fresh* names, that is, all output prefixes are now of the form $(\nu y)(\overline{x}\langle y\rangle.P \mid Q)$. This implies that output-guarded choice is impossible and that each name has a single destination: once sent, it cannot be retransmitted to another location. In Iπ, each transmitted channel is shared between at most two locations.

As we will see later, this calculus re-appears naturally in some typed settings.

The localised, internal mobility LπI-calculus

This calculus was considered by Merro [MS04]. It combines the two previously described restrictions, which ensures that every channel is known to exactly two locations and that only one of them is allowed to use it for input. This is useful when one wants to describe disconnected topologies with explicit addresses, because it disallows to treat channels as globally shared resources.

These restrictions shrink the set of behaviours of individual processes by reducing the amount of contextual information needed to consider. This allows for more localised reasoning about process terms, making this set of restrictions appealing when considering typed variants of π-calculus.

Polyadic π-calculus

The polyadic π-calculus *extends* the monadic π-calculus with polyadic input and output actions. That is, output and input now permits the communication of a vector of names $\vec{d} = d_1, d_2, \ldots, d_n$.

$$\overline{x}\langle\vec{d}\rangle.P \mid x(\vec{y}).Q \longrightarrow P \mid Q\{d_1/y_1, d_2/y_2, \ldots\}$$

In a synchronous setting the polyadic π-calculus can be encoded by simply transmitting each of the names in sequence over a private channel.

1.6 Relative expressiveness of π-calculi: a quest for order

We only considered various purely syntactic restrictions of π-calculus, however we are already in a situation where the variety of options becomes frightening. Syntactically, they can be considered as a sequence of sub-calculi:

$$L\pi I \subset \{L\pi, \pi I\} \subset \pi_a \subset \pi_{nc} \subset \pi_{ic} \subset \pi_{sc} \subset \pi_{mc} \subset \pi_{fc}$$

Nevertheless, this does not immediately imply anything about either the relative expressive power of these languages or their suitability for modelling different distributed systems. For example, as we already discussed, the theory of π_a is quite different from calculi further to the right.

Moreover, we have presented the sub-calculi as incremental restrictions, however many of the restrictions (e.g. asynchrony, locality, mixed-choice) can instead be considered as orthogonal *conditions*. Considering their combinations would

then lead to many more new variants. It is therefore a natural question of both mathematical and practical significance on how these flavours of π-calculus are related, and in particular, whether we can establish a precise classification of these calculi, preferably of a hierarchical or lattice-like order.

A natural way to relate a collection of formal systems is to compare their *expressive power*. As we went from π_{fc} to LπI, we have (gradually) removed two sources of expressiveness: choice and synchronisation (i.e. output prefixing). Is it possible to restore them?

1.6.1 Restoring synchronisation

It is well known both in distributed systems and in process calculus research communities that synchronous communication can be restored via so-called asynchronous *rendezvous* or handshake protocol. Let $\overline{x}\langle\!\langle y \rangle\!\rangle$ and $x(\!(y)\!)$ denote some unspecified encodings of synchronous input and output prefixes in the setting of π_a. There are at least two standard encodings for these operators.

The first is due to Boudol:

Example 1.6.1 (Boudol-style encoding of synchronisation [Bou92])**.**

$$\overline{c}\langle\!\langle d \rangle\!\rangle.P := (\nu r)(\overline{c}\langle r\rangle \mid r(v).(\overline{v}\langle d\rangle \mid P)) \tag{1.6.1}$$
$$c(\!(y)\!).Q := c(r).(\nu v)(\overline{r}\langle v\rangle \mid v(y).Q) \tag{1.6.2}$$

The second is even simpler and is due to Honda and Tokoro:

Example 1.6.2 (Honda-Tokoro-style encoding of synchronisation [HT91])**.**

$$\overline{c}\langle\!\langle d \rangle\!\rangle.P := c(r).(\overline{r}\langle d\rangle \mid P) \tag{1.6.3}$$
$$c(\!(y)\!).Q := (\nu r)(\overline{c}\langle r\rangle \mid r(y).Q) \tag{1.6.4}$$

where it is assumed that the extra names introduced in the encodings (r and v) are temporary and are therefore not free in P. Both of these encodings can be extended to achieve an encoding of π_{nc} into π_a. Namely, we can define a translation function $[\![\cdot]\!] : \pi_{nc} \to \pi_a$, which is homomorphic on all operators except for input and output prefixes, where it is defined as

$$[\![x(y).P]\!] = x(\!(y)\!).[\![P]\!]$$

It is easy to verify that the composition $\overline{x}\langle\!\langle d \rangle\!\rangle.P \mid x(\!(y)\!).Q$ will reduce similarly to the standard synchronous [Comm] reduction. For example, assuming the encoding in in Example 1.6.2, we get that

$$\begin{array}{rcl} \overline{x}\langle\!\langle d \rangle\!\rangle.P \mid x(\!(y)\!).Q & = & x(c).(\overline{c}\langle d\rangle \mid P) \mid (\nu r)(\overline{x}\langle r\rangle \mid r(y).Q) \\ & \xrightarrow{\tau} & (\nu r)(\overline{r}\langle d\rangle \mid P \mid r(y).Q) \end{array}$$

By the assumption that $r \notin \mathsf{fn}(P)$

$$\begin{array}{rcl} & \equiv & (\nu r)(\overline{r}\langle d\rangle \mid r(y).Q) \mid P \\ & \xrightarrow{\tau} & P \mid Q\{d/y\} \end{array}$$

which clearly simulates the [Comm] reduction. The encoding works because the handshake protocol happens over a fresh channel r, which means it is not

observable by any context. The only observable actions are the initial input and output transitions:

$$x(c).(\overline{c}\langle d\rangle \mid P) \xrightarrow{x(r)} \overline{r}\langle d\rangle \mid P \text{ and } (\nu r)\overline{x}\langle r\rangle.(r(d).P) \xrightarrow{(\nu r)\overline{x}\langle r\rangle} (r(d).P)$$

This showcases the usage of scope restriction to achieve process abstraction: the actions introduced by the encoding are internal and are therefore not exposed in any context. Finally, we note that both of these encodings work equally well in the Lπ and Iπ calculi[6], and hence even in the LπI calculus.

1.6.2 Restoring (input-guarded) choice

Input-guarded choice can also be restored as shown by Nestmann and Pierce [NP00]. The idea is to run the possible choice branches in parallel and simulate a transaction protocol to achieve mutual exclusion. We present the encoding of the binary case, but it can easily be generalised to n-ary choice:

Example 1.6.3 (Input-guarded choice encoding)**.**

$$\begin{aligned}[\![c_1(x).P_1 + c_2(x).P_2]\!] \quad &:= (\nu l)(\overline{l}\langle \mathsf{True}\rangle \mid \\ &\quad c_1(x).l(b).(\ \texttt{if}\ b\ \texttt{then}\ [\![P_1]\!]\ \texttt{else}\ \overline{c_1}\langle x\rangle \mid \overline{l}\langle \mathsf{False}\rangle) \mid \\ &\quad c_2(x).l(b).(\ \texttt{if}\ b\ \texttt{then}\ [\![P_2]\!]\ \texttt{else}\ \overline{c_2}\langle x\rangle \mid \overline{l}\langle \mathsf{False}\rangle) \\ &\quad)\end{aligned}$$

In the above example the syntax of π-calculus is extended with booleans and conditionals, however they can also be encoded in π_a and are used only to simplify the presentation. Note that this encoding does not rely on output-prefixing and is also valid Lπ. However, it is not valid in Iπ, because the re-transmitted name x in $\overline{c_i}\langle x\rangle$ is not fresh.

The correctness of the encoding can be sketched by analysing three possible cases:

1. There exists only a message output on the channel c_1, in which case the parallel component prefixed by $c_1(x)$ is evaluated, which simply leads to the execution of the choice branch P_1.
2. Symmetrically for the case of a message only on the channel c_2.
3. If the context contains messages for both channels c_1 and c_2, the two parallel components (which encode the two choice branches), can simultaneously receive the messages on both channels, however they will have to *compete* for the single message True output initially on the "lock channel" l. Eventually, one of the branches wins and proceeds with its continuation, while also informing the other branch via the lock channel that it has failed to acquire the lock, which forces the branch to *retransmit* the message it held.

The crucial property of π_a that this encoding relies on is the congruence in Equation 1.5.1. In particular, the branch which fails to acquire the lock will

[6]Though in the Iπ calculus we only consider the transmission of fresh names

contain a subsequence of this sort

$$c_i(x).(\cdots).\overline{c_i}\langle x\rangle$$

which by the aforementioned congruence has no overall effect. Intuitively, π_a allows to *rollback* any input action simply by retransmitting messages, which is impossible with synchronous interaction, because the sender immediately observes the input transition of the receiving process.

Having restored both output-prefixing and input-guarded choice, can we combine them to recover stronger variants of choice? Before we can answer this question, we need to commit to more formal criteria required for an encoding.

1.6.3 Relative expressiveness and good encodings

When speaking about encodings between process calculi, we implicitly assume that they are in some sense *reasonable*. The exact definition of this criterion very much depends on the motives for considering the relationships between the process calculi in question.

We can identify at least two cases for such comparisons. Firstly, one might be interested in applying results proved for one calculus in another, and therefore wants to transfer or compare their (equational) theories. Secondly, one could be considering implementability of some features and therefore wants to examine their relative expressiveness based on some operational criteria. Naturally, the optimal situation is where an encoding achieves a relative expressiveness result while preserving the equational theory of the source calculus. Moreover, in order to classify all the process calculi, a definite set of criteria should be set to verify the validity of encodings.

Despite the fact that this problem is widely discussed in the literature, no universally agreed-on measure for what constitutes an acceptable encoding of process calculi exists.

Historically the most commonly considered requirement is the so-called *full abstraction*:

Definition 1.6.1 (Full abstraction)**.** An encoding $[\![\cdot]\!]\colon \mathcal{S} \to \mathcal{T}$ from a source process calculus $\mathcal{S}$ to a target process calculus $\mathcal{T}$ is said to be *fully abstract* with respect to a source and a target behavioural equivalence, $\asymp_{\mathcal{S}}$ and $\asymp_{\mathcal{T}}$ respectively, if

$$P \asymp_{\mathcal{S}} Q \text{ iff } [\![P]\!] \asymp_{\mathcal{T}} [\![Q]\!]$$

The *if* part can be understood as a soundness requirement (often called *adequacy* in π-calculus literature), while the other direction as a completeness one.

Assuming that the equivalences are in fact congruences, we can view the full abstraction condition from a purely algebraic perspective: a fully abstract encoding is simply a map between two quotient algebras. It is then immediately clear that this is not a sufficient general condition, because the map can still be completely arbitrary.

In fact, in recent years the requirement of fully abstract encodings has been put into question by several authors. Parrow [Par14], Gorla and Nestmann [GN14] have argued extensively that full abstraction, considered in isolation, can be both too strong and too weak and is therefore not a compulsory requirement for encodings.

As a motivating (non-)example, consider the simple case of comparing the relative expressiveness of π_a and π_{nc}. In Example 1.6.1 and Example 1.6.2 we have presented what would seem like a reasonable encodings of π_{nc} into π_a. However, full abstraction fails for both of these encodings under most combinations of behavioural equivalences which could be considered adequate.

To see this, let $\psi = \overline{x}\langle y\rangle$, then $P = \psi.\psi$ is equivalent to $Q = \psi \mid \psi$ under any behavioural equivalence which ignores causality (this includes all the ones we have defined so far). However, their encodings, $[\![P]\!]$ and $[\![Q]\!]$, into π_a are not behaviourally equivalent under any acceptable relation which is closed under parallel contexts (so not even necessarily a congruence). This can be seen by considering the context $C = [\cdot] \mid x(z)$. $C[[\![P]\!]]$ then has a τ-transition into a stuck process

$$\begin{array}{lcl} C[[\![P]\!]] & = & \overline{x}\langle\!\langle y\rangle\!\rangle.\overline{x}\langle\!\langle y\rangle\!\rangle \mid x(z) \\ & \xrightarrow{\tau} & (\nu c)(c(r).(\overline{r}\langle y\rangle \mid \overline{x}\langle\!\langle y\rangle\!\rangle) \mid \mathsf{end}\{y/z\}) \\ & \equiv & (\nu c)(c(r).(\overline{r}\langle y\rangle \mid \overline{x}\langle\!\langle y\rangle\!\rangle)) \\ & \not\xrightarrow{\mu} & \end{array}$$

while $C[[\![Q]\!]]$ could still perform transitions in the other parallel component.

This failure is due to the fact that the term $x(z)$ is not a result of a translation of any π_{nc} term. While this is not automatically a problem (for example the term $\varepsilon(x)$ is also not a translation of any term, but cannot have any effect due to the conruence in Equation 1.5.1), the term $x(z)$ is said to "not follow the protocol" of the encoding, because after intercepting the private channel, it does not use it in the way the encoding intended. In other words, the embedding induces a "synchronous" sub-algebra of π_a, which is not fully compatible with the asynchronous contexts of π_a. Therefore, the encoding can only be shown sound, but not complete.

Due to similar reasons, the identity embedding of π_a into π_{nc} defined by a homomorphic extension of

$$[\![\overline{x}\langle y\rangle]\!] = \overline{x}\langle y\rangle.\mathsf{end}$$

is also not fully abstract under most natural equivalence choices. This can be seen by observing that $\varepsilon(a) \cong_a^c \mathsf{end}$, but $[\![\varepsilon(a)]\!] = \varepsilon(a) \not\sim \mathsf{end}$.

The above examples are classified as "false negatives" in [GN14]. Full abstraction can also produce "false positives". For example, while the encoding of input-guarded choice in [NP00] is shown to be fully abstract under natural equivalences, it is considered unsatisfactory, because the protocol put forward in the encoding introduces divergence.

Obviously, these examples do not imply that the encodings are "incorrect" or that full abstraction is not a useful notion. In the "false negative" examples,

the lack of completeness implies that we cannot transfer the equational theory (induced by the source equivalence) of the source calculus to its embedding. On the other hand, the "false positive" example shows that a complete encoding can be misleading, because it might be exploiting some unjustifiable behaviours, such as infinite reductions.

In an attempt to resolve this situation, a set of criteria for what constitutes a "good" encoding for comparing relative expressiveness was set forth by Gorla [Gor10].

Gorla replaces the full abstraction condition, which is also often called *observational correspondence*, with a different requirement of *operational correspondence*:

Definition 1.6.2 (Operational correspondence)**.** Let $[\![\cdot]\!] \colon \mathcal{S} \to \mathcal{T}$ be an encoding between two process calculi. We say that $[\![\cdot]\!]$ is *operationally corresponding* if the following two conditions hold:

Operational completeness: for all $P \in \mathcal{S}$,
$P \Longrightarrow P'$ implies $[\![P]\!] \Longrightarrow \asymp_{\mathcal{T}} [\![P']\!]$

Operational soundness: for all $[\![S]\!] \Longrightarrow T$, there exists an S', such that
$S \Longrightarrow S'$ and $T \Longrightarrow \asymp_{\mathcal{T}} [\![S']\!]$.

On the other hand, Gorla introduces additional restrictions, which were previously either assumed implicitly or ignored, which attempt to formally capture the intuitive notion of a reasonable encoding. We will list only one of them which will be relevant for us later:

Definition 1.6.3 (Compositionality)**.** Let $[\![\cdot]\!] \colon \mathcal{S} \to \mathcal{T}$ be an encoding between two process calculi. We say that $[\![\cdot]\!]$ is (weakly) *compositional*, if for every k-ary operator op of $\mathcal{S}$ and for every subset of names N, there exists a k-hole context C^N_{op}, such that for all $S_1, \ldots, S_k$, with $\mathsf{fn}(S_1) \cup \ldots \mathsf{fn}(S_k) = N$, it holds that

$$[\![\mathsf{op}(S_1, \ldots, S_k)]\!] = C^N_{\mathsf{op}}([\![S_1]\!], \ldots, [\![S_k]\!])$$

Since their proposal, Gorla's criteria have been well-received by process algebraists and are being incorporated into many recent works. Gorla's criteria are further abstracted and slightly relaxed by Van Glabbeek in [Gla12], arguing that they rule out some completely reasonable encodings and are slightly too specific. As it is noted in [Gla12], it remains open whether, in general, one set of criteria is preferable to the other. For an excellent survey on possible choices of criteria for "good" encodings we refer the reader to Kirstin Peters' PhD thesis [Pet12].

Throughout the rest of the thesis we will embrace Gorla's criteria and refer to encodings satisfying it as "good".

1.6.4 Wrapping it up

We can now go back to comparing the expressiveness of π-calculi described previously.

The encodings presented previously can all be shown to satisfy Gorla's criteria and can therefore be considered good encodings representing the relative expressiveness. In particular, it is indeed possible to combine them to show that π_a is as expressive as π_{ic}. In fact, Nestmann [Nes00] showed that separate-choice can also be reasonably encoded in π_a while preserving soundness.

However, Gorla's criteria still do not provide us with a definite measure of expressiveness. In particular, it can be shown that there exists a "good" encoding from π_{mc} to π_{sc} ([Nes00]), which would contradict Palamidessi's celebrated separation result in [Pal03], which claims that no reasonable encoding of mixed-choice into π_{sc} exists.

This disagreement is due to the fact that Palamidessi requires the parallel composition operator to be encoded homomorphically, a condition also known as *strong* compositionality, whereas Gorla only requires *weak* compositionally. Gorla observes that while under many types of process equivalences the strong compositionally requirement is not required for proving separation results, it is indeed a necessity[7] when considering various *weak* behavioural equivalences [Gor10, Section 5.1.3].

Although strong compositionally has been argued by Van Glabbeek [Gla12] and others to be an extreme restriction, we consider it to be a much better alternative to weak compositionally, especially when discussing distributed implementations of π-calculi. In later sections (subsection 2.3), we will discuss one more criterion for encodings – preservation of *distributability* – which can be seen as a middle-ground between strong and weak compositionality and has the benefit of not being a (purely) syntactic criterion.

The choice between weak and strong compositionality thus determines whether the hierarchy of expressiveness collapses from π_{mc} to π_a or whether we have a separation result, where π_{mc} is strictly more powerful than π_{sc}, which is equally expressive to π_a.

It is informative to look at Palamidessi's proof to understand why, when considering applications in distributed systems programming, the separation is quite natural.

The separation proof is based on the observation that mixed choice allows two identical processes to *atomically* agree on a conditional branch. This can be exploited to directly solve a form of distributed consensus – a notoriously hard problem in general. In the terminology of Palamidessi, mixed choice allows to "break symmetries". In particular, if we consider a process $P_{xy} = x().\overline{z}\langle x\rangle + \overline{y}\langle\rangle$ and its α-equivalent "mirror" $P_{yx} := P_{xy}\{y/x, x/y\}$, then the composition

$$P_{xy} \mid P_{yx}$$

can be seen as an execution of a leader-election protocol, where both processes P_{xy}, P_{yx} are trying to propose "their" channel (x and y respectively) to some external entity z. The parallel composition can transition into either $\overline{z}\langle y\rangle \mid \mathsf{end}$ or $\mathsf{end} \mid \overline{z}\langle x\rangle$, which can be seen as x or y being chosen as the leader.

[7] Alternatively, Gorla shows that one can consider a less general and slightly ad-hoc form of operational correspondence, however this sacrifices the generality of the criteria.

Separate choice does not possess such capabilities. Consider for example

$$\overline{x}\langle\rangle + \overline{y}\langle\rangle \mid x() + y()$$

While there is still agreement on both sides whether to choose the channel x or y, the processes are *already* asymmetric, which can be understood as some a priori (shared) knowledge. This idea was originally developed in the setting of CSP by Bougé in [Bou88] and is the precursor of Palamidessi's result.

2 (In search of) a distributable π-calculus

In the previous section we discussed some general properties and relative expressiveness results of the π-calculus. We now assume a much more focused view and attempt to identify a (minimal) sub-calculus, which could be used as a formal basis for a distributed programming language for distributed programming. In doing so, we re-discover another relevant criterion for comparing π-calculi.

As we saw earlier, the various π-calculi form a rich, but complicated family of process calculi. When considering applications, it is clearly beneficial to pick the smallest or minimal language from an expressiveness class that satisfies the application needs. In the context of programming languages, this means that we can keep the amount of built-in or primitive constructions minimal and treat the encodings of features of the extended calculi (from the same expressiveness class) as regular programs. This not only eases implementation, but, more importantly, it reduces the amount of effort needed in order to perform formal verification and static analysis of programs.

However, as we saw in subsubsection 1.6.4, there is no universal measure of expressiveness for π-calculi, therefore it is hard to pin-point the ideal sub-calculus for any given setting without assuming some subjective measure. Our subjective measure will be the feasibility of a *fully distributed* implementation of the calculus in question. In this respect our concerns will, in many ways, coincide with those of Fournet and Gonthier in their work on the Join-calculus [FG00] and, to a lesser extent, to Merro and Sangiorgi's work on the (L)πI calculus [Mer00] and Pierce's work on the Pict programming language [PT00].

2.1 Revisiting the building blocks

In this subsection we describe our choices on the semantics of the core composite parts which make up a process calculus. Additionally, we identify further conditions which we argue are necessary for an implementation to be realizable in a fully distributed manner.

While these options can be viewed and considered in isolation, it is exactly the interplay between them that allows us to combine them in a coherent way. In particular, our intuition for what constitutes implementable features is based on the idea that parallel sub-terms of π-calculus should not share any common state or knowledge. This implies that interaction can be implemented as a pure

message passing system and no coordination is needed to execute processes in parallel. These goals subsume the ideas of *localised* channels, appearing in the Join-calculus and Lπ, and πI.

2.1.1 Overall model: synchrony versus asynchrony

The decision whether to assume a synchronous or asynchronous model of interaction requires almost no thought in our setting. We have seen that the asynchronous π-calculus is sufficiently expressive to encode all reasonable primitives of the extended calculi. In particular, features not expressible in π_a are also not feasible in a practical implementation.

More concretely, π-calculus with asynchronous communication can be implemented very straightforwardly by realizing output actions as concrete messages being transfered between processes. This is not the case with synchronous communication. In fact, one can analyse the situation from a distributed knowledge (see [HM00]) perspective for better insight. In the synchronous case, interaction achieves, atomically, common knowledge between two processes. From the reduction $\overline{x}\langle y\rangle.P \mid x(z).Q \longrightarrow P \mid Q\{y/z\} = Q'$ we can deduce that

- $\phi_0 :=$ "Q' knows y"
- $\phi_1 :=$ "P knows that ϕ_0"
- $\phi_2 :=$ "Q' knows that ϕ_1"
- $\phi_3 :=$ "P knows that ϕ_2"
- ...

by virtue of synchronisation. However, the asynchronous encoding of synchronous interaction represents only the first two of iterations of ϕ_i. Recall the reduction from earlier:

$$\begin{array}{rcl} P \mid Q = \overline{x}\langle\!\langle d\rangle\!\rangle.P' \mid x(\!(y)\!).Q' & = & x(c).(\overline{c}\langle d\rangle \mid P') \mid (\nu r)(\overline{x}\langle r\rangle \mid r(y).Q') \\ & \longrightarrow & (\nu r)(\overline{r}\langle d\rangle \mid r(y).Q') \mid P' \\ & \longrightarrow & P' \mid Q'\{d/y\} \end{array}$$

After the first reduction we only have that $\psi_0 :=$ "P knows r" holds, however it does not hold that $\psi_1 :=$ "Q knows that ψ_0", because we assume that inputs are not observable in an asynchronous setting. After the second reduction, by receiving on r, Q learns ψ_0 and d. However, once again P' does not know ψ_1 and "Q knows d", it can only postulate that "Q *will* know d" by assuming eventual message delivery. We can see from this example that in order to achieve an identical level knowledge in the asynchronous case, we would have to iterate the synchronisation encoding indefinitely – an impossible feat.

This is in tune with results from distributed systems research. In particular, the well-known Two Generals Problem [Gra78, p. 465] can be reformulated to say that it is impossible to achieve common knowledge in finite time in the presence of arbitrary message delivery delays.

In fact, these considerations imply it is only reasonable to speak about *reliable* implementations of such "full" synchronisation primitives in localised settings,

where an arbiter can coordinate two otherwise independent processes. Such settings include shared-memory or shared-processor models, where two concurrent processes can synchronise via a shared resource.

As a side remark, it is interesting to note that these differences are not apparent when we consider properties of π-calculi using standard tools. It would be therefore an interesting direction for future work to examine properties of π-calculi and their encodings from the perspective of distributed (epistemic) knowledge. For example, Palamidessi's separation result discussed previously also originates from considerations about distributed consensus.

2.1.2 Message delivery semantics

In subsubsection 1.2.1 we discussed that the choice between synchronous and asynchronous message delivery is not binary, but actually a range of options. Moreover, we observed that one of these options – FIFO-ordered delivery – can be seen as optimal in terms of balance between strength and realizability.

However, neither FIFO-ordered, nor causally-ordered delivery resurfaced in our earlier discussion of relative expressiveness of π-calculi. One could infer that the collapse from π_{sc} to π_a (subsubsection 1.6.4) implies that the intermediate degrees of asynchrony are expressible in π_a and therefore, do not need to be considered.

The actual situation is quite different. In fact, we are not aware of any "good" encodings of FIFO-ordered communication in the π-calculus. In particular, all of the potential encodings are not compositional.

It is not clear how to even express the condition of FIFO-ordered delivery in the π-calculus. This is mainly due to the fact every π-calculus process is (potentially) a composition of other parallel processes and therefore it is not possible to infer, by design, what are its sequential execution steps. As noted in [PSN11], there are two ways in which causal dependencies are introduced in the π-calculus: nesting of prefixes and nesting of scope restriction operators. We have seen examples of using such dependencies in encodings (Example 1.6.1, Example 1.6.2, Example 1.6.3).

We would wish to formally define when a encoding can be said to preserve FIFO-ordered communication semantics, however as discussed in [PSN11, Sec. 2.4], it is not clear how this can be done in general way. We therefore, unfortunately, have to rely on intuition.

As usual, FIFO-delivery can be simulated by intermediate buffer processes, however this is not acceptable for our purposes. For the simplified case where each channel is only shared between two processes, a "continuation-passing style" encoding into the polyadic variant of π_a is proposed in [DeY+12]:

Example 2.1.1 (FIFO-order encoding for binary communications)**.**

$$[\![\overline{c}\langle x\rangle.P]\!] = (\nu c')(\overline{c}\langle x, c'\rangle \mid [\![P\{c'/c\}]\!])$$
$$[\![c(y).P]\!] = c(y, c').[\![P\{c'/c\}]\!]$$

Each output action also transmits a fresh continuation channel, which is used both by the sender and by the receiver in their respective continuations. This ensures that the receiver can only receive messages in a fixed order by accessing their continuations. We note that this encoding fails in the presence of self-communication, for example in the encoding of

$$[\![\overline{c}\langle x\rangle.c(y)]\!] = (\nu c')(\overline{c}\langle x, c'\rangle \mid c'(y, c''))$$

the output is offered for channel c, while the continuation expects input on c'.

It is possible to extend this encoding to at least handle multiple senders, by using the following modified encoding of input:

$$[\![c(y).P]\!] = c(y, c').(!c(x, k).\overline{c'}\langle x, k\rangle \mid [\![P\{c'/c\}]\!])$$

Here, a forwarding process is inserted, which "remembers" the original channel name and forwards messages from it to the latest continuation. Note that this encoding can also accommodate self-communication.

Even ignoring the fact that both of these encodings are not compositional, they are still quite unsatisfactory: the first encoding is overly restrictive, while the second encoding introduces a new replicated process for *each* message received, which hints at severe engineering headaches.

All of these complications leave us with no option, but to work with a formulation of π-calculus which assumes FIFO-buffered communication as primitive. As noted before, the implementation of FIFO-buffered delivery poses few problems in practice. Additionally, we will see later that FIFO-buffered semantics are very natural for applying Session Type systems. Some implications of this choice will be discussed after we present FIFO-buffered semantics in later sections.

2.1.3 Channel locality and ownership

Perhaps the main source of issues when considering the π-calculus as a distributed pure message-passing system is its treatment of names. Although we often refer to π-calculus names as channels, it can be argued that considering π-calculus names as channels quickly leads to confusion.

The main problem is that, intuitively, we might understand a channel as some kind of connection between two locations, however this interpretation is not valid for the π-calculus. This situation is observed by Lévy ([Lév97]):

> As a general observation, it is very difficult to implement in a distributed fashion languages based on CCS or the pi-calculus, even in their asynchronous versions. The main obstacle is that a channel resides in the ether, and therefore is not located. If a message is sent to an unlocated port, one comes very quickly to solve a distributed consensus for nearly every communication, since two receptors on a same channel have to agree in order to take the value and thus to prevent the other from getting the same message.

The consensus problem Lévy refers is that given a term such as

$$P \mid Q \mid Z = c(x).P' \mid c(y).Q' \mid \overline{c}\langle z \rangle$$

and assuming all 3 parallel sub-terms exist as processes in different locations, in order to *route* the output action either to P or Q, there must exist either a buffer shared by P and Q or an intermediate coordinating process, or alternatively P and Q must directly coordinate to decide which is allowed to consumed the message.

Several approaches are known in the literature for resolving this problem. Perhaps the most impressive attempt is Fournet and Gonthier's Join-calculus, which can be obtained as "an 'extended subset' of the asynchronous π-calculus by amalgamating the three operators for input, restriction, and replication into a single operator, called *definition*, but with the additional capability to describe the atomic *joint* reception of values from two different channels." ([Nes98]). Since the reduction semantics of Join-calculus is quite different from that of π-calculus, the discussion of it is outside of our scope. However, we note that names in the join-calculus have two unique properties:

Locality Names have a 'fixed location' associated to them.

Uniformity Input names are unique and replicated. Message reception is deterministic and is more similar to function application.

The Lπ calculus also deals with channel locality problems. As mentioned in the introduction, Lπ prohibits received names to appear as input subjects in the continuation. This can be interpreted as a model where names can be treated as shared entities *locally*, but not remotely. For example, in the term

$$(\nu c)(P \mid Q \mid R) \mid Z := (\nu c)(c(x).P' \mid c(y).Q' \mid \overline{z}\langle c \rangle) \mid z(a).\overline{a}\langle \mathbf{1} \rangle.Z'$$

Z can receive the channel c, but it can only use it for output, therefore the channel remains local to the sub-term $(\nu c)(P \mid Q \mid R)$. However, the sub-terms P and Q are permitted to concurrently read from the channel because it is local to both of them and can be assumed to be shared. This term would also be valid in LπI.

Nevertheless, we believe this restriction is still not sufficient, because it implies a non-uniform treatment of processes: the implementation must distinguish between local and non-local communication. Modern distributed systems using computers with many-core architectures exhibit self-similarity, in the sense that local computation, all the way down to the hardware layer, is becoming more and more similar to the message-passing based 'remote' distributed computation happening over the network [Bau+09]. Therefore it does not seem beneficial to distinguish between local and non-local computation as it can only impede the capabilities for process mobility.

Instead, we propose to abolish the idea that two parallel entities can perform reception on the same name. We call this condition *full ownership*, implying that channels are always owned by a particular process. This idea has been previously considered (with very similar motivation) by Amadio in his work on the π_1 calculus [Ama00]:

> [...] we assume that every channel name is associated with a *unique* process which receives messages addressed to that name (communication becomes point-to-point). To emphasize the unicity of the receptor, we will refer to the calculus as the π_1-calculus. We note that asynchronous point-to-point communication does not require synchronizations between possibly distant processes and therefore it makes minimal assumptions on the capabilities of the distributed system.

Amadio uses a linear typing discipline to enforce the 'unicity' of input prefixing, however he additionally requires that input is persistent. In particular, simple terms such as $c(x).\mathsf{end}$ are not considered well-typed. The type system also does not impose the locality principle of Lπ, so a received name name can still be used for input.

We take a different approach and extend the output-capability constraint of Lπ with another syntactic restriction:

Definition 2.1.1 (Full ownership restriction)**.** For any name c, and any process P such that $P = (\nu\vec{n})(P_1 \mid \ldots \mid P_n)$, c appears as a (free) input subject in at most one parallel-component P_i.

This restriction can be formulated syntactically only because of the presence of the Lπ restriction, which prevents aliasing problems. We will elaborate on this when we formally define all the constraints.

It is not clear whether this restriction reduces the expressive power of Lπ. It is known that Lπ can be encoded into the similarly restricted calculus π_1 [Ama00], so we have reason to believe that expressiveness is not lost.

More importantly, we sketch encodings which show that there exist "reasonable" encodings of primitives, which are normally expressed using concurrent input from a shared channel. For example, we can show that the encoding of input-guarded choice (Example 1.6.3) can be modified to not rely on reading from a shared channel:

Example 2.1.2 (Encoding of input-guarded choice for owned channels)**.** The modification replaces the shared 'lock channel', with a process that provides a 'lock service':

$$\begin{aligned}
&\mathrm{LockService}(l) := l(r).(\overline{r}\langle\mathsf{True}\rangle \mid l(r').\overline{r'}\langle\mathsf{False}\rangle)\\
&\mathrm{ReadLock}(l,b) := (\nu v)(\overline{l}\langle v\rangle \mid v(b))
\end{aligned}$$

$$\begin{aligned}
[\![c_1(x).P_1 + c_2(x).P_2]\!] := (\nu l)(&\mathrm{LockService}(l) \mid\\
&c_1(x).\mathrm{ReadLock}(l,b).(\texttt{if } b \texttt{ then } [\![P_1]\!] \texttt{ else } \overline{c_1}\langle x\rangle) \mid\\
&c_2(x).\mathrm{ReadLock}(l,b).(\texttt{if } b \texttt{ then } [\![P_2]\!] \texttt{ else } \overline{c_2}\langle x\rangle)\\
&)
\end{aligned}$$

In the n-ary case the lock service should continue outputting False on every received channel.

Another important example is encoding internal or non-deterministic choice,

which, without is usually expressed as

$$P \sqcap Q := \tau.P + \tau.Q \approx (\nu i)(\bar{i}\langle \mathbf{1} \rangle \mid i().P \mid i().Q)$$

but is illegal under the full ownership restrictions. However, we can easily modify it to avoid reading on a shared channel by introducing a degree of indirection:

Example 2.1.3 (Encoding of non-deterministic choice for owned channels)**.**

$$P \sqcap Q := (\nu c, l, r)(\overline{c}\langle l \rangle \mid \overline{c}\langle r \rangle \mid l().P \mid r().Q \mid c(z).\overline{z}\langle \mathbf{1} \rangle)$$

This encoding works because it is not determined which of the channels, l or r, will reach c first.

We therefore believe that the locality and full ownership conditions are not overly restrictive and keep our calculus sufficiently expressive. In addition to simplified implementation, we note that these restrictions also eliminate a large class of hazardous behaviours, such as accidental aliasing or deadlocks which most commonly occur when there are multiple consumers of a shared resource.

2.1.4 Choice

We have seen that all the reasonable flavours of choice can be encoded into π_a and that these encodings are also compatible with locality and full ownership restrictions.

Unfortunately, the encoding of input-guarded choice (Example 1.6.3) is not compatible with FIFO-ordered message delivery. This can be seen by considering the following process

$$\bar{l}\langle a \rangle \mid \overline{r}\langle b_1 \rangle.\overline{r}\langle b_2 \rangle \mid [\![l(x).r(y_1).r(y_2) + r(x).P_2]\!]$$

It is possible that, at first messages a and b_1 are communicated over channels l and r and, furthermore, the branch for channel l is chosen. This means that the message b_1 received on the other branch has to be re-transmitted, i.e. the process above reduces to

$$\overline{r}\langle b_2 \rangle \mid r(y_1).r(y_2) \mid \overline{r}\langle b_1 \rangle$$

and it is now clearly possible that the messages b_i are received out of order.

On the other hand, assuming locality and full ownership, the above situation can never occur (using the encoding in Example 2.1.2), because in the encoded process the name r would appear in two parallel components. However, this then implies that all of the process sums are "trivial" in the sense that they need to completely split up the owned channels.

We therefore choose to re-introduce input guarded choice as a primitive. The full ownership restriction naturally extends to input-guarded sums. Moreover, mostly as a convenience and a matter of hygiene, we require that channel names

appearing as subjects of the input guards are distinct, so as to separate internal choice from external.

Including input-guarded choice as a primitive is justified from the implementation point of view, because the full-ownership restriction implies that all the channels appearing as subjects of the input-guards are indeed local to the process. Since they are local, the process can inspect the respective channel buffers and receive from one as soon as it finds a non-empty one. In fact, because we do not assume causally-ordered delivery, it can even perform the inspection in a straightforward *round-robin* manner without violating message delivery semantics or even fairness assumptions.

We do not consider stronger kinds of choice, because, like many things we have ruled out already, they are not realisable in a distributed setting without performing consensus or introducing mediators. In fact, at this point we can claim that, *if* there were a realizable encoding of, for example, separate choice, then it should be expressible under the given constraints we have identified.

2.1.5 Summary

We have identified the following set of constraints and features which we believe could form a basis for a fully *distributable* flavour of π-calculus:

- *Asynchronous communication over FIFO buffered channels.* Asynchronous in this case means that output is non-blocking.
- *Localised channels*, permitting to only send the output capability of a name, in the style of Lπ-calculus.
- *Full ownership* which further restricts the above to disallow an input prefix to simultaneously appear in two (possibly) parallel processes.
- *Input-guarded choice* as an explicit primitive to allow more convenient forms of control flow.
- *Disjoint sums*, to eliminate non-deterministic choice from sums and thus simplify reduction analysis.

In the next section, we will materialize these choices in the form of a (severely restricted) π-calculus, which we call $\pi_{\texttt{dist}}$.

We should also mention that we have chosen to exclude process replication, mostly to simplify the presentation, but also because in future work it would be more interesting to examine the addition of explicit (co-)recursion.

2.2 $\pi_{\texttt{dist}}$, formally

We now present the $\pi_{\texttt{dist}}$ calculus formally.

2.2.1 Base syntax and its restrictions

The syntax of $\pi_{\texttt{dist}}$ is essentially the same as of the synchronous π-calculus with input-guarded choice, π_{ic}, except that we exclude replication. The structural congruence rules also remain exactly the same as in Figure 2, except that some of them no longer apply because they concern non-existent terms (e.g. the congruence for replication).

One can split the syntax of $\pi_{\texttt{dist}}$ into two layers: the base 'user' syntax and the runtime syntax which extends the base syntax with channel queue processes. We postpone the definition of the runtime syntax until the next subsection.

Definition 2.2.1 ((User) syntax of $\pi_{\texttt{dist}}$)**.** The $\pi_{\texttt{dist}}$ user syntax is defined by the following grammar

$$\begin{array}{rcll} P, Q, R & ::= & (\nu c)P & \text{name restriction} \\ & | & \overline{c}\langle n\rangle.P & \text{output} \\ & | & \sum_{i=1}^{n} c_i(n).P_i & \text{(disjoint) input-guarded choice} \\ & | & P \mid Q & \text{parallel composition} \\ & | & \mathsf{end} & \text{empty process} \end{array}$$

with the following (syntactic) restrictions:

(Local channels) For every term $c(n).P$, the (free) name n does not appear as an input subject in P.

(Full ownership) For any term $P = (P_1 \mid P_2 \mid \dots P_k)$, and any (free) name n, n appears as input subject in at most one parallel component P_i.

(Disjoint sums) All channel names c_i are distinct in input-guarded choice expressions $\sum_i c_i(x_i).P_i$.

We additionally define the regular input prefix as a one element sum

$$c(d).P := \sum_{i=1}^{1} c(d).P$$

and retain the use of $(+)$ for an associative commutative binary operator for sums, i.e.

$$c_1(d_1).P_1 + c_2(d_2).P_2 := \sum_{i=1}^{2} c_i(d_i).P_i$$

A couple of remarks should be made about this definition.

First of all, note that if we ignored the extra syntactic restrictions and re-used the standard reduction semantics of π-calculus, we would just get π_{ic}.

Secondly, it is not immediately obvious that the syntactic restrictions faithfully implement the intended constraints described in subsubsection 2.1.5. In particular, it is not obvious that these restrictions can be (reasonably) expressed using syntactic means. The reason the syntactic restrictions work is because the combination of the rules avoids potential *aliasing* problems. More concretely, reduction will not result in violations of the syntactic restrictions. This is achieved

by the (Local channels) rule, which ensures that no channel aliases are used for input. Moreover, it is precisely because of this rule that the (Full ownership) and (Disjoint sums) restrictions can be formulated using syntactic means.

As a counter example, if the (Local channels) rule was not present, we could form the term

$$x(y).(a(z) \mid y(z)) \xrightarrow{x(a)} (a(z) \mid a(z))$$

which renders the (Full ownership) rule useless.

Similarly,

$$x(y).(a(z) + y(z)) \xrightarrow{x(a)} (a(z) + a(z))$$

shows that without the (Local channels) rule, the (Disjoint sums) restriction is also meaningless.

Note that these restrictions do not prevent *self-communication*, i.e. performing both input and output on the same channel. This is both not desired and impossible to enforce syntactically. For example:

$$x(y).\overline{y}\langle z\rangle \xrightarrow{x(x)} \overline{x}\langle z\rangle$$

2.2.2 Semantics and FIFO-buffered communication

The user syntax presented above enforces all the restrictions we described in subsection 2.1 except for the asynchronous FIFO buffered channel semantics. In fact, as already remarked in the previous section, equipping the syntax with standard π-calculus reduction rules would result in a restricted variant of π_{ic}, which is a synchronous calculus. This is due to the presence of output-prefixing.

There are at least two possible ways to establish FIFO ordering.

One is to extend the process terms (and syntax) with special buffer processes, which act as mediators between channel output and input and modify the reduction rules to ensure communication always happens via the buffer processes. This is very much in the spirit of the classical encodings of asynchronous communication into π-calculus. Multiple variations of such representations exist in the literature (see [BPV08][DeY+12][HYC08]).

Another approach is to keep the usual syntax of the synchronous π-calculus, but parameterise all reduction and transitions over a "global" channel buffer context, which tracks the contents of channel buffers. This approach is taken in [Den+13].

There are certain trade-offs between the two approaches. The benefit of the first approach is that channel queue processes appear at the object level and therefore one can use the standard π-calculus tools and techniques to reason about the process terms. However, it can be rather inconvenient because process terms are no longer freely generated and one needs to take care to distinguish between terms which contains queues and those which are "plain". This implies that, for example, achieving relative expressiveness results is quite complicated, because one needs to partition the process terms into certain classes.

On the other hand, a parameterised representation of buffers such as in [Den+13] suffers from the problem that all of the derivations and reductions become *contextual*. Moreover, there are problems with scope restriction rules, as noted in [Den+13, p. 9], where it is resorted to ad-hoc scope "extension" rules. While it might be possible to apply the technique of tracking name usage via a context as it is done in the LTS described in [Sew00], this still means that one needs to consider a context of channel buffers which is "global" and thus makes it harder to reason about scope restriction in the natural way. Nevertheless, it could be argued that this approach is cleaner, because it clearly separates the representation of process structure and operational semantics. In particular, processes are freely generated.

Observe that the syntactic restrictions of $\pi_{\texttt{dist}}$ already imply that $\pi_{\texttt{dist}}$ processes are not freely generated. Moreover, we believe that the benefits of having channel queues at the object level outweighs the burden of treating them with some extra special care. Additionally, this avoids parameterisation of the transition system and therefore we choose the first approach to make $\pi_{\texttt{dist}}$ channels FIFO-buffered.

We extend the base syntax of $\pi_{\texttt{dist}}$ (Definition 2.2.1) with explicit channel queue processes.

Definition 2.2.2 (Channel queue process syntax and related notions)**.** A channel queue for the channel c, with a message buffer (i.e. vector of names) $\vec{d}$ is denoted by $c{:}[\vec{d}]$.

We use $[\,]$ to represent the empty buffer (i.e. an empty queue), $[m, \vec{d}]$ to represent a buffer with the name m at the beginning (i.e. m will be next element to be dequeued from the queue) and $[\vec{d}, m]$ to represent a buffer where m was the last enqueued element.

Moreover, we extend the definition of free and bound names in Definition 1.4.2 to include the channel queue processes. All names in buffers are considered free, including the channel name, therefore:

$$\mathsf{names}(c{:}[\vec{d}]) = \mathsf{fn}(c{:}[\vec{d}]) = \{c\} \cup \{d_1, \ldots, d_n\} \qquad \mathsf{bn}(c{:}[\vec{d}]) = \emptyset$$

We note that ν binds queue names as usual, so in $x{:}[] \mid (\nu y)y{:}[\vec{d}]$, x is a free name, but y is not.

Finally, to maintain the intended FIFO semantics and ensure that for each channel there is at most one channel queue process, we impose the restriction that a queue process for the channel c can appear at most once within a term and either as a parallel sub-term or when it is empty and is guarded by a scope restriction for the name c. That means that a queue can only occur in two kinds of process terms: $P \mid c{:}[\vec{d}]$ or $C[(\nu c)(P \mid c{:}[])]$ for some context C.

For example, this is not a legal term: $c{:}[] \mid c{:}[]$. Neither is $\alpha.(c{:}[\vec{d}] \mid P)$.

The last restriction might seem rather complicated and ad-hoc, but the intuition behind it is simple: we wish to introduce process queues (only) in tandem with name restriction, because we will interpret name restriction as channel *creation*. The situation will become more clear in later sections, but for now observe that

Figure 5 Reduction rules of $\pi_{\texttt{dist}}$

$$\begin{array}{rcll}
\overline{c}\langle d\rangle.P \mid c{:}[\vec{m}] & \longrightarrow & P \mid c{:}[\vec{m}, d] & [\textsc{Send}] \\
\sum_{i=1}^{n} c_i(x).P_i \mid c_k{:}[d, \vec{m}] & \longrightarrow & P_k\{d/x\} \mid c_k{:}[\vec{m}], 1 \leq k \leq n & [\textsc{Recv}] \\
P \longrightarrow P' & \text{implies} & (\nu c)P \longrightarrow (\nu c)P' & [\textsc{Scop}] \\
P \longrightarrow P' & \text{implies} & P \mid Q \longrightarrow P' \mid Q & [\textsc{Par}] \\
P \equiv P', Q \equiv Q' \text{ and } P' \longrightarrow Q' & \text{implies} & P \longrightarrow Q & [\textsc{Str}]
\end{array}$$

terms such as

$$x(y).(x{:}[] \mid P)$$

have no reasonable interpretation, because the queue for the name x appears after the name is used as a prefix subject.

Finally, while we wish to be able to discuss 'open' terms, we intend to only introduce queues in tandem with name restriction, i.e. as closed terms:

$$(\nu c)(P \mid c{:}[])$$

Therefore a queue can never be guarded by input or output prefixes, even when it appears as a parallel sub-term. This would be an illegal term: $x(y).(c{:}[] \mid P)$.

These restrictions might seem rather cumbersome, but they are straightforward to apply in practice and will be abstracted away once we introduce a small programming language which decodes to the $\pi_{\texttt{dist}}$ syntax.

These definitions will be useful later:

Definition 2.2.3 (Queue-open and queue-closed terms)**.** We say that a process term P is c-closed if $c \in \mathsf{fn}(P)$ and there exists a channel queue c as a parallel component of P, i.e. $P \equiv (\nu \vec{s})(P_1 \mid \cdots \mid P_n \mid c{:}[\vec{d}])$. (Note that $c \in \mathsf{fn}(P)$ implies $c \notin \vec{n}$.) Otherwise, if $c \in \mathsf{fn}(P)$, but the latter condition fails, then we say that P is c-open.

We use the general term queue-open and queue-closed if want to speak about closure with respect to an arbitrary queue.

Definition 2.2.4 (Owned channels/names)**.** For a process P, we defined the set of its owned channels (or names), $\mathsf{owned\text{-}n}(P)$, as all the input-prefix subject names which are free in P.

For example, $\mathsf{owned\text{-}n}(c_1(x).(\nu c)(c_2(y).\overline{m}\langle r\rangle \mid c(z))) = \{c_1, c_2\}$

2.2.3 Reduction semantics

The reduction semantics of $\pi_{\texttt{dist}}$ are given in Figure 5. They differ from the usual π_{fc} rules (Figure 3) only in that the rule [Tau] is removed and the rule [Comm] is replaced by two rules [Send] and [Recv], which now mediate the communication via a buffer. Note that it is implicitly assumed that this buffer always exists in parallel to the sending process and that there is always a *unique* buffer process per name. This means that the application of [Send] is always

possible on an output-prefixed term. Moreover, this ensures the FIFO ordering semantics, as the buffer causes multiple outputs from a single process to be serialised.

2.2.4 Structural congruence

We also have to extend the structural congruences of π_{fc} (Figure 2) by the following rule in order to enable channel "garbage collection":

$$(\nu c)c{:}[] \equiv \mathsf{end}$$

This is a standard technique for ensuring that we can throw out terms which can no longer be used (see e.g. the added structural rules for discarding unused conditional branches when encoding booleans in [NP00, A.1]). As an example of how this works, consider this reduction (here we assume $c \notin \mathsf{fn}(P) \cup \mathsf{fn}(Q)$):

$$\begin{array}{rcl}
(\nu c)(\overline{c}\langle d\rangle.P \mid c(m).Q \mid c{:}[]) & \longrightarrow & (\nu c)(P \mid c(m).Q \mid c{:}[m]) \\
& \longrightarrow & (\nu c)(P \mid (Q\{m/x\} \mid c{:}[]) \\
& \equiv & P \mid (\nu c)(Q\{m/x\} \mid c{:}[]) \\
& \equiv & P \mid Q\{m/x\} \mid (\nu c)c{:}[] \\
& \equiv & P \mid Q\{m/x\} \mid \mathsf{end} \\
& \equiv & P \mid Q\{m/x\}
\end{array}$$

After the channel c is used to send the message from P to Q, it is no longer used in either of the processes and hence its queue can also be discarded. Note that while we could also add another structural rule which would discard bound queues that are *not* empty, this would potentially hide the fact that some processes are sending messages which are never being received, so we do not incorporate such a rule.

In [BPV08], another structural rule of the form $P \equiv P \mid c{:}[]$ is added. The rule is said to "only apply" when P does not already contain some queue. We do not use this rule as it implies that we either permit duplicate channel buffers (which then violates FIFO semantics) or $\equiv$ is no longer a congruence, both of which are unacceptable. As a counterexample, let $P = \overline{c}\langle z\rangle$ and $Q = c(z)$, then $P \equiv P \mid c{:}[] =: P'$ and $Q \equiv Q \mid c{:}[] =: Q'$, however $P \mid Q \not\equiv P' \mid Q'$.

What *can* be added is the congruence

$$(\nu c)P \equiv (\nu c)(P \mid c{:}[])$$

This ensures that as long as we are working with closed terms it is guaranteed that there is a unique channel queue associated with every channel name. Intuitively it can be understood as saying: whenever you create a new channel name, also create its associated queue. As mentioned earlier, our intention is to introduce queues only in tandem with name restriction. We therefore incorporate this rule as part of the congruence rules for $\pi_{\mathtt{dist}}$. Later, we will show that it fits in well with the reduction and transitions semantics of $\pi_{\mathtt{dist}}$.

It should be noted that it can only be applied when P is a c-open term, because otherwise we get an invalid term. For example, this is not allowed: $(\nu c)(P \mid c{:}[]) \not\equiv (\nu c)(P \mid c{:}[] \mid c{:}[])$.

We summarise these considerations with the following definition

Definition 2.2.5 (Structural rules of $\pi_{\texttt{dist}}$)**.** The structural rules of $\pi_{\texttt{dist}}$ consist of the standard structural rules of π-calculus (Figure 2), except for replication (there is no replication in $\pi_{\texttt{dist}}$) and with the addition of the following two rules:

$$(\nu c)(c{:}[]) \equiv \mathsf{end}$$
$$(\nu c)P \equiv (\nu c)(P \mid c{:}[])$$

2.2.5 Actions and transition rules

In order to properly treat buffers and obtain the LTS for $\pi_{\texttt{dist}}$, we have to slightly modify and extend the LTS and actions of π_{fc}. As already discussed in the previous subsection, there are different ways to formulate FIFO-buffered communication. Here, we take a different approach from both [Den+13] and [BPV08]. The reason for doing so, is because we want to be able to discuss the translations of both *queue-open* and *queue-closed* (to be defined) terms and relate those of the former and those of the latter. For this reason, we explicitly label buffer actions and take care to maintain *harmony* with the reduction semantics.

Definition 2.2.6 ($\pi_{\texttt{dist}}$ actions)**.** We extend the set of actions with

$$\mu' := \mu \mid \alpha'^{b}$$

which are used to denote buffer actions. The free and bound name functions are defined identically as in the case of unannotated, or *raw*, actions. Additionally, we use the notation $\mu^A, \mu^B, \ldots$ to denote possibly-annotated actions, i.e. $A, B, \ldots$ can be either 'blank' or b. It is implicitly assumed that the annotated action μ^A is not τ. Finally, we reserve the usage of μ for raw actions.

Figure 6 presents the LTS for $\pi_{\texttt{dist}}$. The rules are almost the same as the ones for π_f, except that we now also have to account for the buffers. The rules OutQ, InpQ describe the channel queue behaviour. Moreover, we extend the rule Open to account for buffer actions and we impose additional side conditions to the Close and Comm rules to ensure that communication always happens via a buffer (i.e. we prevent processes communicating directly), respecting FIFO semantics. We also replace the simple input rule Inp with input-guarded choice.

Finally, we split the Par into 3 separate rules, which might look quite intimidating and therefore require some explanation. First of all, observe that removing the extra side-conditions and collapsing the different rules into one, would produce the original Par rule. These additional constraints are there in order to prevent nonsensical transitions, based on whether a channel queue related to the current action is present in the derived process.

More concretely, for each input or output subject name a appearing in a process term P, it can either be *open* or *closed* with respect to the channel queue of a. We want to be able to consider terms open and closed with respect to the queues-involved. In fact, using this formulation we are able to consider actions of queues in isolation and both when they *close* a channel-owning process and a process which produces output into that channel.

As an example, the application of Par should not be allowed, when one process performs a raw action, while there is a queue which closes it under that action

Figure 6 Transition rules of $\pi_{\mathtt{dist}}$

$$\frac{}{\overline{x}\langle y\rangle.P \xrightarrow{\overline{x}\langle y\rangle} P}\ (\mathsf{Out}) \qquad \frac{1 \le k \le n}{\sum_{i=1}^{n} c_i(x).P_i \xrightarrow{c_k(z)} P_k\{z/x\}}\ (\mathsf{Inp})$$

$$\frac{}{c{:}[\vec{d}] \xrightarrow{c(m)^b} c{:}[\vec{d}, m]}\ (\mathsf{InpQ}) \qquad \frac{}{c{:}[m, \vec{d}] \xrightarrow{\overline{c}\langle m\rangle^b} c{:}[\vec{d}]}\ (\mathsf{OutQ})$$

$$\frac{P \xrightarrow{\mu} P' \qquad \mathsf{bn}(\mu) \cap \mathsf{fn}(Q) = \emptyset \qquad Q \text{ is } \mathsf{subj}(\mu)\text{-open}}{P \mid Q \xrightarrow{\mu} P' \mid Q}\ (\mathsf{Par\text{-}Raw})$$

$$\frac{P \xrightarrow{x(y)^b} P'}{P \mid Q \xrightarrow{x(y)^b} P' \mid Q}\ (\mathsf{Par\text{-}BufInp})$$

$$\frac{\mu^b \text{ is an output action} \qquad \begin{array}{c} P \xrightarrow{\mu^b} P' \\ \mathsf{bn}(\mu) \cap \mathsf{fn}(Q) = \emptyset \end{array} \qquad \mathsf{subj}(\mu) \notin \mathsf{owned\text{-}n}(Q)}{P \mid Q \xrightarrow{\mu^b} P' \mid Q}\ (\mathsf{Par\text{-}BufOut})$$

$$\frac{P \xrightarrow{\overline{x}\langle y\rangle^A} P' \qquad y \neq x}{(\nu y)P \xrightarrow{(\nu y)\overline{x}\langle y\rangle^A} P'}\ (\mathsf{Open})$$

$$\frac{P \xrightarrow{(\nu z)\overline{x}\langle z\rangle^A} P' \qquad Q \xrightarrow{x(z)^B} Q' \qquad z \notin \mathsf{fn}(Q) \qquad A \neq B}{P \mid Q \xrightarrow{\tau} \nu z(P' \mid Q')}\ (\mathsf{Close})$$

$$\frac{P \xrightarrow{\overline{x}\langle y\rangle^A} P' \qquad Q \xrightarrow{x(y)^B} Q' \qquad A \neq B}{P \mid Q \xrightarrow{\tau} P' \mid Q'}\ (\mathsf{Comm})$$

$$\frac{P \xrightarrow{\mu'} P' \qquad z \notin \mathsf{names}(\mu')}{(\nu z)P \xrightarrow{\mu'} (\nu z)P'}\ (\mathsf{Res})$$

$$\frac{P \equiv P' \qquad P \xrightarrow{\mu'} Q \qquad Q \equiv Q'}{P' \xrightarrow{\mu'} Q'}\ (\mathsf{Cong})$$

in parallel. In theory, we could ignore this side-condition, however this would permit 'trash' processes which exhibit 'unrealisable' transitions. Consider the transition

$$x(y).P \xrightarrow{x(z)} P\{z/y\} =: P'$$

If we had the usual Par rule, we could apply it to obtain

$$\frac{x(y).P \xrightarrow{x(z)} P'}{x(y).P \mid x{:}[] \xrightarrow{x(z)} P' \mid x{:}[]} \quad (?\ \mathsf{Par})$$

However, there is no valid $\pi_{\mathtt{dist}}$ context in which this transition could lead to a *reduction*, because it would require a dual queue action (and there can be no duplicate queues for the same channel). The problem here is that a raw action on channel x, which should – when "closed up" by the x queue – induce a communication between the x-open process and the queue for x resulting in a τ-transition, "escapes" the queue closure. The situation is identical for raw output, as captured by the Par-Raw rule.

The situation is slightly different with buffer-actions. This is due to the fact that there can be only one channel queue and a single consumer for it, but there can be many producers. Therefore we don't need to imply any side-conditions on Par-BufInp, because a channel queue *input* action *can* commute past whatever it is enclosing. For example, the following transition derivation is valid:

$$\text{(Par-BufInp)}\ \frac{\dfrac{x{:}[] \xrightarrow{x(y)^b} x{:}[y]}{\overline{x}\langle z\rangle.P \mid x{:}[] \xrightarrow{x(y)^b} \overline{x}\langle z\rangle.P \mid x{:}[y]} \qquad \dfrac{}{\overline{x}\langle y\rangle.Q \xrightarrow{\overline{x}\langle y\rangle} Q}}{\overline{x}\langle z\rangle.P \mid x{:}[] \mid \overline{x}\langle y\rangle.Q \xrightarrow{\tau} \overline{x}\langle z\rangle.P \mid x{:}[y] \mid Q}\ (\mathsf{Comm})$$

However, the situation is different for buffer-output, consider

$$\frac{x{:}[y,\vec{d}] \xrightarrow{\overline{x}\langle y\rangle^b} x{:}[\vec{d}]}{x(y).P \mid x{:}[y,\vec{d}] \xrightarrow{\overline{x}\langle y\rangle^b} x(y).P \mid x{:}[\vec{d}]} \quad (?\ \mathsf{Par})$$

Again this transition cannot be matched by a dual action, because it requires a parallel process that performs input on x, which is not allowed in $\pi_{\mathtt{dist}}$.

To build up our intuition about $\pi_{\mathtt{dist}}$, we now prove a couple of propositions which characterise the transitions of $\pi_{\mathtt{dist}}$ processes.

Proposition 2.2.1 (Raw-action induced normal forms)**.** *For all $P \in \pi_{\mathtt{dist}}$:*

1. *If $\alpha = \overline{x}\langle y\rangle$ or $\alpha = x(y)$ then $P \xrightarrow{\alpha} P'$ implies that P is x-open and there exist Q, R and $\vec{n}$, such that: $P \equiv (\nu\vec{n})(\alpha.Q + (\sum_i S_i) \mid R)$, $P' \equiv (\nu\vec{n})(Q \mid R)$, where $\vec{n} \cap \mathsf{names}(\alpha) = \emptyset$.*
2. *Moreover, if α is an input action, then also $\mathsf{subj}(\alpha) \notin \mathsf{owned\text{-}n}(R)$ and $\mathsf{subj}(\alpha) \in \mathsf{owned\text{-}n}(R)$.*
3. *Similarly, if we consider $\alpha = (\nu y)\overline{x}\langle y\rangle$, then the previous result holds, except that the restriction for $\vec{n}$ is flipped and becomes $y \in \vec{n}$.*

Proof.

1. By induction on the derivations.

   By case analysis of the LTS rules, $P \xrightarrow{\alpha} P'$ implies that the derivation for this transition begins with either Out or Inp. WLOG, assume that α is an output action and that the base case is an application of Out to achieve $O + (\sum_i S_i) \xrightarrow{\alpha} O'$. The base case satisfies the criteria.

   Now, assume we have a derivation such that the conclusion is $P \xrightarrow{\alpha} P'$ and the induction hypothesis holds.

   Again by case analysis, only the following rules can use the premise to create a new process with the same transition: Par-Raw, Res and Cong.

   If the rule is Par-Raw, then the side-condition implies that Q is $\mathsf{subj}(\alpha)$-open and therefore $P \mid Q$ is $\mathsf{subj}(\alpha)$-open as well and the shape matches.

   The other two rules also do not affect the shape and or queue-openness, therefore by induction we are done.

2. Immediate from the previous part by the full-ownership restriction.

3. Again by induction on the derivations. We can consider the same derivations as in the first part, except that now they will contain also contain an application of Open. The Open rule does not change structure and so the result follows.

□

Proposition 2.2.2 (Buffer-action induced normal forms)**.** *For all $P \in \pi_{\mathtt{dist}}$:*

1. *If $\alpha = \overline{x}\langle y\rangle^b$ or $\alpha = x(y)^b$ then $P \xrightarrow{\alpha} P'$ implies that P is x-closed and for some R and buffer contents $\vec{d}$, $\vec{d'}$:*

$$P \equiv (\nu \vec{n})(x{:}[\vec{d}] \mid R)$$
$$P' \equiv (\nu \vec{n})(x{:}[\vec{d'}] \mid R)$$

   where $\vec{n} \cap \mathsf{names}(\alpha) = \emptyset$.

   Moreover, if $\alpha = \overline{x}\langle y\rangle^b$, then $\vec{d} = [y, \vec{d'}]$ and $x \notin \mathsf{owned\text{-}n}(R)$.

   Otherwise, if $\alpha = x(y)^b$ is an input action, then $\vec{d'} = [\vec{d}, y]$.

2. *Similarly, if we consider $\alpha = (\nu y)\overline{x}\langle y\rangle^b$, then the previous result holds, except that the restriction for $\vec{n}$ is flipped and becomes $y \in \vec{n}$.*

Proof. Similarly to the previous proposition. □

Proposition 2.2.3 (Buffer transitions)**.** *For all $P \in \pi_{\mathtt{dist}}$:*

1. *$P \xrightarrow{x(y)^b}$ iff P is x-closed. (That is: an x-closed process is always input enabled on x).*

2. *P is x-closed does* not *imply $P \xrightarrow{\overline{x}\langle y\rangle^b}$.*

Proof. 1. Directly follows from the definition of channel queue transitions.

2. If the queue for x is empty, then it simply cannot make any transitions.

□

2.2.6 Harmony of the LTS

At the very least, an LTS should be in agreement with the reduction semantics and structural properties of a process calculus.

Lemma 2.2.1 (Harmony of $\pi_{\texttt{dist}}$ LTS)**.** *The LTS of $\pi_{\texttt{dist}}$ satisfies the conditions in Lemma 1.4.1.*

Proof. **Congruences.** Since we include the Cong rule in our LTS, the $\equiv$-parts of the proof are achieved for free. However, it is good to check whether the congruences are still in tune with the LTS.

If we excluded the Cong rule and instead used double left-right rules for the Par-*, Close and Comm rules, we would only have to worry about the non-standard structural rules $(\nu c)c{:}[] \equiv \mathsf{end}$ and $(\nu c)P \equiv (\nu c)(P \mid c{:}[])$

For the first congruence, both end and $(\nu c)c{:}[]$ are unable to perform any transitions (and therefore affect reductions), so we only need to consider processes which transition or reduce *to* them. However, this means we only need to consider the second part of the lemma, and we also get the congruence part for free by the [Str] reduction rule.

The situation is different with the second congruence. Consider

$$(\nu c)\overline{c}\langle y\rangle.P \equiv (\nu c)(\overline{c}\langle y\rangle.P \mid c{:}[]) \xrightarrow{\tau} (\nu c)(P \mid c{:}[y])$$

however $(\nu c)(\overline{c}\langle y\rangle.P)$ is not able to make any transition at all, failing the harmony rule. We therefore would need to extend the LTS with a rule such as

$$\frac{(\nu c)(P \mid c{:}[]) \xrightarrow{\tau} P'}{(\nu c)P \xrightarrow{\tau} P'} \quad (\nu\text{-queue-contract})$$

Note that we only need cater for the τ-transitions, because the congruence only affects actions related to c and those are hidden by the name restriction. Since $(\nu c)(P \mid c{:}[])$ can perform at least the reductions of $(\nu c)P$, with the above rule added all the τ-transitions of the two congruent terms coincide and hence congruence problems are resolved.

Reductions.

The more important part of the lemma is showing that τ-actions coincide precisely with reduction steps (second clause of the lemma). This ensures that our LTS is modelling the same reductions we have specified in the reduction rules.

The proof is done by tedious case analysis of the reduction and transition rules, so we only sketch the main clauses.

The if part: First of all, the [Send] and [Recv] reduction rules are faithfully represented by the derivation

$$\dfrac{\dfrac{}{\overline{c}\langle m\rangle.P \xrightarrow{\overline{c}\langle m\rangle} P}\ (\mathsf{Out}) \qquad \dfrac{}{c{:}[\vec{d}] \xrightarrow{c(m)^b} c{:}[\vec{d}, m]}\ (\mathsf{InpQ})}{\overline{c}\langle m\rangle.P \mid c{:}[\vec{d}] \xrightarrow{\tau} P \mid c{:}[\vec{d}, m]}\ (\mathsf{Comm})$$

and

$$\dfrac{\dfrac{}{Q + c(x).P \xrightarrow{c(m)} P\{m/x\}}\ (\mathsf{Inp}) \qquad \dfrac{}{c{:}[m, \vec{d}] \xrightarrow{\overline{c}\langle m\rangle^b} c{:}[\vec{d}]}\ (\mathsf{OutQ})}{Q + c(x).P \mid c{:}[m, \vec{d}] \xrightarrow{\tau} P\{m/x\} \mid c{:}[\vec{d}]}\ (\mathsf{Comm})$$

respectively.

The only if part: There are only two rules from which a τ-action transition can appear: Comm and Close. For Comm, it is easy to see that the only possible instantiation of it is a derivation which has one of the two shapes above (in the proof of the *if* part). This is a consequence of lemmas 2.2.1 and 2.2.2.

The case of Close is a little bit more tricky, but if we examine the LTS rules, we notice that even though the action labels are duplicated into raw and buffered, the open and closure rules and the restriction on Par-BufOut is exactly the same as in the standard LTS formulation for π-calculus and therefore the semantics of "hiding" is unchanged. The only thing which is different is that the application of Open and Close rules now happens two times, instead of one (as in the classical π-calculus). This is because the channel queue processes are internalised and at first the scope is extended to the buffer and only later to the receiver. As an example, consider

$$\dfrac{\dfrac{\overline{x}\langle y\rangle.P \xrightarrow{\overline{x}\langle y\rangle} P}{(\nu y)\overline{x}\langle y\rangle.P \xrightarrow{(\nu y)\overline{x}\langle y\rangle} P}\ (\mathsf{Open}) \qquad \dfrac{}{x{:}[] \xrightarrow{x(y)^b} x{:}[y]}\ (\mathsf{InpQ})}{(\nu y)\overline{x}\langle y\rangle.P \mid x{:}[] \xrightarrow{\tau} (\nu y)(P \mid x{:}[y])}\ (\mathsf{Close})$$

and then

$$(\textsc{Open})\ \dfrac{\dfrac{(P \mid x{:}[y]) \xrightarrow{\overline{x}\langle y\rangle^b} (P \mid x{:}[])}{(\nu y)(P \mid x{:}[y]) \xrightarrow{(\nu y)\overline{x}\langle y\rangle^b} P \mid x{:}[]} \qquad \dfrac{}{x(z).Q \xrightarrow{x(y)} Q\{y/z\}}\ (\mathsf{Inp})}{(\nu y)(P \mid x{:}[y]) \mid x(z).Q \xrightarrow{\tau} (\nu y)(P \mid x{:}[] \mid Q\{y/z\})}\ (\mathsf{Close})$$

So we have that

$$(\nu y)\overline{x}\langle y\rangle.P \mid x{:}[] \mid x(z).Q \xrightarrow{\tau} (\nu y)(P \mid x{:}[y]) \mid x(z).Q \xrightarrow{\tau} (\nu y)(P \mid x{:}[] \mid Q\{y/z\})$$

as expected.

□

2.2.7 Incomplete and complete terms

The main reason for introducing the annotated buffer actions is because we want to be able to abstract out away the buffer, which can often be seen just as a communication medium, and only consider the communication *content*. In particular, we wish to discuss the connection between buffer and non-buffer actions of processes.

Proposition 2.2.4. *For all $\pi_{\mathtt{dist}}$ processes P, it does not hold that both $P \xrightarrow{\alpha}$ and $P \xrightarrow{\alpha^b}$, where $\alpha = \mu \setminus \{\tau\}$.*

Proof. Since $P \xrightarrow{\alpha^B}$, $\mathsf{subj}(\alpha) \in \mathsf{fn}(P)$, but then P is either queue-closed or queue-open and therefore by Proposition 2.2.1 and Proposition 2.2.2 the result follows. □

Proposition 2.2.5 (Queue extraction and queue closure)**.** *For every x-open process P and every reduction sequence $P \stackrel{\alpha_1}{\Longrightarrow}\stackrel{\alpha_2}{\Longrightarrow} \cdots \stackrel{\alpha_n}{\Longrightarrow} P'$, where $\mathsf{subj}(\alpha_i) = x$, there exist queue processes $Q = x{:}[\vec{d}]$ and $Q' = x{:}[\vec{d'}]$, such that $P \mid Q \Longrightarrow P' \mid Q'$.*

Proof. Let (β_i) be the subsequence of input actions of (α_i). Respectively, let (γ_i) be the subsequence of output actions. Then define

$$\vec{d} := \mathsf{obj}(\beta_1), \mathsf{obj}(\beta_2), \ldots, \mathsf{obj}(\beta_k)$$

and

$$\vec{d'} := \mathsf{obj}(\gamma_1), \mathsf{obj}(\gamma_2), \ldots, \mathsf{obj}(\gamma_m)$$

The proof follows. □

Note however that this proof relies on early transition semantics and results in a very non-optimal queue.

For example, for the process $\overline{x}\langle y\rangle.x(y)$ it will infer the queues $\vec{d} = \vec{d'} = [y]$. However, an empty buffer would have sufficed.

2.3 On distributability and the expressive power of $\pi_{\mathtt{dist}}$

We now shortly return to comparing π-calculi. We have already remarked on some aspects of the relative expressiveness of $\pi_{\mathtt{dist}}$ as compared to other π-calculi, in particular that the expressiveness of $\pi_{\mathtt{dist}}$ is at most that of $(\mathrm{L})\pi_a$. However, we now argue that the full ownership restriction separates $\pi_{\mathtt{dist}}$ from $\mathrm{L}\pi_a$ in terms of the degree of *distributability*.

Informally, the distributability of a calculus represents the degree up to which process terms can be executed independently. Distributability arises quite naturally when considering semantically defined (weaker) alternatives of the strong compositionality requirement. As already noted, forcing encodings to translate the parallel operator homomorphically can be seen as too harsh. For example, the following (weakly) compositional encoding would be invalid:

$$[\![P \mid Q]\!] = (\nu x, y)(\overline{x}\langle\rangle \mid \overline{y}\langle\rangle \mid x().[\![Q]\!] \mid y().[\![P]\!])$$

Nevertheless, it can be argued that the context introduced is completely *local* and that it does not impede possibilities for parallel execution. On the other hand, there are clearly bad weakly compositional translations of the parallel operator (for example those that introduce 'broker' processes), which sequentialise the execution of the parallel components by introducing extra causal relationships.

As a middle ground between the two extremes, Peters, Nestmann, and Goltz formally define the notion of distributability and require encodings to preserve the degree of distributability. In [PNG13] they attempt to compare join-calculus with π-calculi, therefore define distributability in a very general way. However, as noted in [PNG13], in the context of a replication-free π-calculus, distributability can be state in a much simpler way:

Definition 2.3.1 (Distributability)**.** A (replication-free) process P is distributable into $P_1, \ldots, P_n$ if $P \equiv (\nu \vec{n})(P_1 \mid \ldots \mid P_n)$ where each $P_i \not\equiv \mathsf{end}$.

For example, given $P = \overline{x}\langle y \rangle$ and $Q = x(y)$, the term $(P \mid P \mid Q)$ can be distributed into the following process groups (up to re-ordering):
$\{(P \mid P), Q\}, \{P, (P \mid Q)\}, \{P, P, Q\}$

We can now define what it means for an encoding to preserve distributability.

Definition 2.3.2 (Preservation of Distributability)**.** An encoding $[\![\cdot]\!] \colon \mathcal{S} \to \mathcal{T}$ preserves distributability if for all source processes $S \in \mathcal{S}$ which are distributable into $S_1, \ldots, S_n \in \mathcal{S}$, there exist $T_1, \ldots, T_n \in \mathcal{T}$, such that they are distributable in $[\![S]\!]$ and $T_i \asymp [\![S_i]\!]$ for all i, where $\asymp$ is (at least) a success-sensitive (weak) reduction bisimulation.

Here success-sensitivity means, informally, that the bisimulation relates only 'successful reductions' of subprocesses, i.e. a terminating subprocess is never related to a diverging one. Preservation of distributability usually does not make sense for encodings which are not "good". In particular, operational correspondence is required, otherwise the comparison of reductions of the source and target calculi is not very meaningful. Moreover, the equivalence $\asymp$ has to be weak, otherwise the translation cannot introduce any extra reductions, therefore rendering the whole notion useless.

We can now sketch a proof that $\pi_{\mathtt{dist}}$ is separated from π_a in terms of distributability. Note that we present this only as an informal proof, because proper comparison of π_a and $\pi_{\mathtt{dist}}$ is quite delicate due to the differences in the formulations of the their respective LTS. We assume existence of an appropriate formulation of weak barbed bisimilarity for $\pi_{\mathtt{dist}}$, which we denote as $\dot{\approx}_d$, although for the sake of the proof we can pretend this is the usual weak barbed bisimilarity.

Proposition 2.3.1. *There is no (good) distributability preserving encoding of π_a into $\pi_{\mathtt{dist}}$.*

Proof. Sketch: Let $[\![\cdot]\!]$ be a good encoding of π_a into $\pi_{\mathtt{dist}}$. Consider the π_a term $S = c(x) \mid c(x)$.

By definition of distributability preservation, we have that $[\![S]\!] \equiv (\nu \vec{n})(T_1 \mid T_2)$ for some T_i, such that $[\![c(x)]\!] \overset{\bullet}{\approx}_d T_i$. Then by the definition of weak barbed bisimilarity, this then implies that both of the T_i are able to perform input actions on c, which implies that c must appear as an input subject in both T_1 and T_2. Since this is not allowed in $\pi_{\texttt{dist}}$ by the full ownership restriction, we arrive at a contradiction. □

As distributability is weaker than strong compositionality, we have the following obvious corollary:

Corollary 2.3.1. *There is no "good" and strongly compositional encoding of π_a into $\pi_{\texttt{dist}}$.*

We note that the above separation result could be strengthened by using a reduction bisimulation instead of barbed bisimilarity. This would slightly complicate the proof because we would have to pick a more complicated source term than S which would actually reduce by performing two parallel reductions on the shared channel. Intuitively it is easy to see why the proof would still work: while in π_a it is possible to perform a sequence of parallel input actions on shared names, any translation of it in π_a would have to introduce a degree of indirection (cf. Example 2.1.3), which serialises access to the shared channels.

The concept of distributability and distributability preservation has surfaced in π-calculus literature only recently ([PN12], [PNG13]). It is a particularly interesting notion for at least two reasons. First of all, it abstractly addresses an important practical concern. Namely, it can be used as a measure to determine whether a particular feature of a process calculi can be implemented inside another calculus, without sacrificing its potential for parallelism. Secondly – and perhaps even more importantly – distributability provides process algebraists with a good criterion for improving the taxonomy of process calculi.

In particular, in [PNG13], Peters, Nestmann, and Goltz present a novel separation result between π_{mc}, π_a and Join-calculus. The separation result strengthens the separation of π_{mc} and π_{sc} proved in [Pal03] by showing that it holds under weaker conditions (specifically, preservation of distributability instead of strong compositionality). More interestingly, the separation of π_a and Join-calculus materialises the intuition that join-calculus describes processes which use *localised* channels and therefore can be executed in a fully distributed environment, unlike processes in π_a.

3 Programming with $\pi_{\texttt{dist}}$

In the previous sections, we have identified a very restrictive subset of π-calculus. Moreover, we have compared its relative expressiveness both in terms of its ability to simulate the usual primitives which come with the asynchronous π-calculus and its degree of distributability. In both cases the results seem to align with our goals to have a minimal process calculus which is sufficiently expressive, but does not constrain distributability.

By design, $\pi_{\texttt{dist}}$ eliminates some types hazardous behaviour, such as race-conditions and deadlocks induced by concurrent reads on a single channel. Nevertheless, processes in $\pi_{\texttt{dist}}$ can still easily exhibit unanticipated non-determinism (Example 2.1.3) or otherwise *go wrong* due to incompatible communication protocols.

We would like to free the programmer who is using $\pi_{\texttt{dist}}$ from such headaches by allowing him to write distributed programs which are safe by construction. To achieve this goal, in this section we attempt to apply the so-called Session Typing discipline to $\pi_{\texttt{dist}}$ in order to restore. To paraphrase the iconic words of Milner, we want "well-typed programs" to "not go wrong".

3.1 A tiny distributed programming language

Although the type system we will be describing later directly assigns types to $\pi_{\texttt{dist}}$ processes, we now make a slight detour and introduce a toy language unimaginatively called TinyPi.

There are two reasons for doing this.

The first reason is that the alternative syntax of TinyPi seems to us slightly more 'natural'. The syntactic restrictions of $\pi_{\texttt{dist}}$ can lead to confusion when considering $\pi_{\texttt{dist}}$ process terms in the syntax of π-calculus.

The second reason is that the new syntax emphasizes channel-ownership and makes the sequential fragments (i.e. isolated processes) 'surface' from $\pi_{\texttt{dist}}$ terms. This view also reveals how the language can be implemented in a straightforward way as a message passing system.

The syntax of TinyPi is given in Figure 7. We will treat $\pi_{\texttt{dist}}$ as the 'assembly' language of TinyPi and provide translational semantics to TinyPi by a decoding of its syntax into the syntax of $\pi_{\texttt{dist}}$.

3.1.1 Translation of TinyPi to $\pi_{\texttt{dist}}$

The translation from TinyPi to $\pi_{\texttt{dist}}$ is relatively straightforward.

Definition 3.1.1 (Translation from TinyPi to $\pi_{\texttt{dist}}$)**.** We define the translation function

$$[\![\cdot]\!] \colon \mathsf{TinyPi} \to \pi_{\texttt{dist}}$$

Figure 7 Syntax of TinyPi

⟨*proc*⟩	::=	⟨*cont-proc*⟩ ‘`,`’ ⟨*proc*⟩	(continuation)
	|	‘`recv`’ ⟨*rec-binder*⟩*	(input clauses)
	|	‘`fresh`’ ⟨*chan*⟩ ‘`in`’ ⟨*proc*⟩	(fresh name generation)
	|	‘`(`’⟨*proc*⟩‘`)`’	(grouping)
	|	‘`END`’	(unit process)
⟨*cont-proc*⟩	::=	⟨*chan*⟩ ‘`!`’ ⟨*var*⟩	(output)
	|	⟨*chan*⟩ ‘`<- spawn`’ ⟨*proc-spec*⟩	(child creation)
⟨*proc-spec*⟩	::=	⟨*chan*⟩ ‘`>-`’ ⟨*proc*⟩	(process specification)
⟨*rec-binder*⟩	::=	‘`|`’ ⟨*chan*⟩‘`(`’⟨*var*⟩‘`) ->`’ ⟨*proc*⟩	
⟨*chan*⟩	::=	⟨*var*⟩	
⟨*var*⟩	::=	⟨*letter*⟩ [⟨*letter*⟩ | ⟨*number*⟩]*	
⟨*letter*⟩	::=	‘`a`’ | ‘`b`’ | ...	
⟨*number*⟩	::=	‘`1`’ | ‘`2`’ | ...	

Notes:

- – `fresh c in P`
  – `recv | k(d) -> P`
  – `c <- spawn Q, P`
  – `d >- P`

  are all binding occurrences of `c` and `d` in `P`.
- We additionally define `c ? \d -> P` as a shorthand for `recv | c(d) -> P`.
- We often omit the continuation `END`.

by the following rules:

$$
\begin{aligned}
[\![\texttt{c >- P}]\!]^n &= [\![\texttt{P}]\!]^n_c := [\![\texttt{P}]\!]\{n/c\} \\
[\![\texttt{n <- spawn Q, P}]\!] &= (\nu n)([\![\texttt{Q}]\!]^n \mid [\![\texttt{P}]\!]), \text{ where} \\
&\qquad \mathsf{fn}([\![\texttt{Q}]\!]^n) \subseteq \{n\} \text{ and } n \notin \mathsf{owned\text{-}n}([\![\texttt{P}]\!]) \\
[\![\texttt{c ! d, P}]\!] &= \overline{c}\langle d\rangle.[\![\texttt{P}]\!] \\
[\![\texttt{recv } \mathit{binders}]\!] &= \sum_{b \in \mathit{binders}} [\![b]\!] \\
[\![\texttt{fresh c in P}]\!] &= (\nu c)([\![\texttt{P}]\!] \mid c{:}[]) \equiv (\nu c)[\![\texttt{P}]\!] \\
[\![\texttt{| c(d) -> P}]\!] &= c(d).[\![\texttt{P}]\!] \\
[\![\texttt{END}]\!] &= \mathsf{end}
\end{aligned}
$$

We consider as valid TinyPi terms only those expressions which translate to valid $\pi_{\texttt{dist}}$ terms, i.e. the domain of the function $[\![\,\cdot\,]\!]$.

The first two rules are the most important ones. The `spawn` operation is a combination of parallel composition and new channel creation. As input, it takes a *process specification* `c >- P`, which defines a process `P`, that is abstracted over its *address* `c`. In $\pi_{\texttt{dist}}$, an address is a channel owned by P, i.e. a name on which the continuation P can perform input actions. Observe that since $(\nu c)P \equiv (\nu c)(P \mid c{:}[])$, a channel buffer process is also 'automatically created'.

The idea behind the `spawn` operator is that, in practice, every new (parallel) processes has to have some place of origin, therefore we represent this explicitly by a *parent* process spawning a *child* process. The side-condition on free names implies that the only name (and in fact, the only 'thing') which is shared between the parent and the spawned process is the address of the child process, which gets bound to the variable `n` in `n <- spawn Q`. Note that the only activity the child is able to perform initially is receiving on its address, because it does not know of anything else.[8]

We observe that these additional restrictions were not present in $\pi_{\texttt{dist}}$, but they do not affect overall expressiveness. We introduced them to keep the presentation as simple as possible and also to show how we can model processes which share the absolute minimum of information.

Additionally, we note that this formulation 'stratifies' the process expressions, i.e. we can now extrapolate sequential execution stories from TinyPi programs. To explain this more concretely, in $\pi_{\texttt{dist}}$, from the expression

$$PQ = (\nu c)(\overline{c}\langle x\rangle.P \mid c(y).Q)$$

we infer that the two processes co-exist in parallel due the presence of the parallel composition operator and only after inspecting their prefixes, we can deduce that there is a causal dependency between them. Namely, the LHS process will have to first execute the output action and only then the RHS process can transition with the receive. In the syntax of TinyPi, the same expression would look like:

[8] Technically, it can also send messages to itself, but that is not very useful.

```
pq = c <- spawn (c' >-
       c' ? \y -> Q'
     ),
     c ! x,
     P
```

In fact, assuming $[\![\texttt{P}]\!] = P$ and $[\![\texttt{Q}]\!]^c_{c'} = Q$, it's translation $[\![\texttt{pq}]\!]$ would produce exactly the $\pi_{\texttt{dist}}$ process above:

$$\begin{aligned}
[\![\texttt{pq}]\!] &= (\nu c)([\![\texttt{c ! x, P}]\!] \mid [\![\texttt{c' >- c' ? \y -> Q}]\!]^c) \\
&= (\nu c)(\overline{c}\langle x\rangle.[\![\texttt{P}]\!] \mid [\![\texttt{c' ? \y -> Q}]\!]^c_{c'}) \\
&= (\nu c)(\overline{c}\langle x\rangle.[\![\texttt{P}]\!] \mid (c'(y).[\![\texttt{Q}]\!])\{c/c'\}) \\
&= (\nu c)(\overline{c}\langle x\rangle.[\![\texttt{P}]\!] \mid c(y).[\![\texttt{Q}]\!]^c_{c'}) \\
&= (\nu c)(\overline{c}\langle x\rangle.P \mid c(y).Q) \\
&= PQ
\end{aligned}$$

We (subjectively) believe that the TinyPi syntax is more intuitive in this example, because its reading implies that in order to perform output on some name, one has to explicitly postulate existence of a process which would receive on it. Additionally, it becomes straightforward to infer which names are owned by which processes.

Despite this 'stratification', the syntax of TinyPi reflects some of the structure of $\pi_{\texttt{dist}}$ operators (Figure 2), as witnessed by the following proposition.

Proposition 3.1.1 (Preservation of structural congruences)**.** *For brevity we define two syntactic macros:*

```
spwnP := a <- spawn P
spwnQ := b <- spawn Q
```

We have that

$$\begin{aligned}
[\![\texttt{spwnP, spwnQ}]\!] &\equiv [\![\texttt{spwnQ, spwnP}]\!] \\
[\![\texttt{spwnP, x <- spawn (y >- END)}]\!] &\equiv [\![\texttt{spwnP}]\!] \\
[\![\texttt{fresh c in END}]\!] &\equiv [\![\texttt{END}]\!] \\
[\![\texttt{fresh x in fresh y in P}]\!] &\equiv [\![\texttt{fresh y in fresh x in P}]\!] \\
[\![\texttt{fresh x in fresh y in P}]\!] &\equiv [\![\texttt{fresh y in fresh x in P}]\!]
\end{aligned}$$

It should be mentioned that in the above proposition there are no equivalents of parallel composition associativity and scope extrusion rules in Figure 2. The latter one becomes redundant, because the spawned children do not share any names. The former does not hold in $\pi_{\texttt{dist}}$, because channels are scoped differently, but this can be fixed by explicitly passing them from the parents.

3.1.2 A distributed "Hello, world!" program

Perhaps due to the constant struggle of taming computers and teaching them to understand humans better, it is a tradition, when presenting new programming

languages, to provide an example program which, when executed, greets the user by displaying the phrase `"Hello, world!"` on the screen.

As our first example, we will also consider greetings, though we will not interfere with the computer and will make two processes greet each other, instead of talking to us.

Below we present a short program. We laxly extend the syntax with process definitions, and, in addition to names, allow text strings to be transmitted.

Listing 1: "Hello, relative!"

```
1  childP = me >-
2      me ? \parent ->
3          parent ! "Hello, parent!",
4          me ? \greeting -> END
5
6  parentP = me >-
7      child <- spawn childP,
8      child ! me,
9      child ! "Hello, child!",
10     me ? \greeting -> END
11
12 mainP = p <- spawn parentP, END
```

The process `mainP` is the top-level process. It spawns the `parentP` process, which in addition spawns the child. Afterwards, it sends to the child his own name, after which they exchange greetings and terminate. To verify this program works as expected, we translate it into a $\pi_{\texttt{dist}}$ process. To save space we denote the greeting strings as `hiP` and `hiC`.

$$
\begin{aligned}
[\![\texttt{childP}]\!]^c &= [\![\texttt{me >- ...}]\!]^c_{me} \\
&= c(\texttt{parent}).[\![\texttt{parent ! hiP,} \\
&\qquad\qquad \texttt{me ? \greeting -> END}]\!]^c_{me} \\
&= c(\texttt{parent}).\overline{\texttt{parent}}\langle\texttt{hiP}\rangle. \\
&\qquad\qquad [\![\texttt{me ? \greeting -> END}]\!]^c_{me} \\
&= c(\texttt{parent}).\overline{\texttt{parent}}\langle\texttt{hiP}\rangle. \\
&\qquad\qquad c(\mathit{greeting}).\mathsf{end}
\end{aligned}
$$

(Note: below we use placeholders `P1, P2, P3` for continuations to save space.)

$$
\begin{aligned}
[\![\texttt{parentP}]\!]^p &= [\![\texttt{child <- spawn childP, P1}]\!]^p_{me} \\
&= (\nu \texttt{child})([\![\texttt{childP}]\!]^{\texttt{child}} \mid [\![\texttt{child ! me, P2}]\!]^p_{me}) \\
&= (\nu \texttt{child})([\![\texttt{childP}]\!]^{\texttt{child}} \mid \\
&\qquad \overline{\texttt{child}}\langle p\rangle.[\![\texttt{child ! hiC, P3}]\!]^p_{me}) \\
&= (\nu \texttt{child})([\![\texttt{childP}]\!]^{\texttt{child}} \mid \\
&\qquad \overline{\texttt{child}}\langle p\rangle.\, \overline{\texttt{child}}\langle \texttt{hiC}\rangle. \\
&\qquad\qquad [\![\texttt{me ? \textbackslash greeting -> END}]\!]^p_{me}) \\
&= (\nu \texttt{child})([\![\texttt{childP}]\!]^{\texttt{child}} \mid \\
&\qquad \overline{\texttt{child}}\langle p\rangle.\, \overline{\texttt{child}}\langle \texttt{hiC}\rangle. \\
&\qquad\qquad p(\texttt{greeting}).\texttt{end})
\end{aligned}
$$

Finally, we compose the two process specifications.

$$
\begin{aligned}
[\![\texttt{main}]\!] &= [\![\texttt{p <- swawn parentP, END}]\!] \\
&= (\nu p)([\![\texttt{parentP}]\!]^p \mid [\![\texttt{END}]\!]) \\
&= (\nu p)([\![\texttt{parentP}]\!]^p \mid \texttt{end}) \\
&\equiv (\nu p)[\![\texttt{parentP}]\!]^p
\end{aligned}
$$

We now inline the definition from above, but rename `child` to c, the two bound `greeting` occurances to g_1 and g_2, and `parent` to x, to save space and to avoid name capture.

$$
\begin{aligned}
&= (\nu p)(\nu c)([\![\texttt{childP}]\!]^c \mid \\
&\qquad \overline{c}\langle p\rangle.\overline{c}\langle \texttt{hiC}\rangle.p(g_1).\texttt{end}) \\
&= (\nu p)(\nu c)(c(x).\overline{x}\langle \texttt{hiP}\rangle.c(g_2).\texttt{end} \mid \\
&\qquad \overline{c}\langle p\rangle.\overline{c}\langle \texttt{hiC}\rangle.p(g_1).\texttt{end})
\end{aligned}
$$

It is easy to see that this communication protocol is *safe*, in the sense that no matter what reduction path it takes, it will always reduce to end and not endure any type-mismatches between the messages transmitted.

Observe that the protocol safety relies critically on FIFO-ordered message delivery semantics. In particular, if we consider the same protocol under synchronous communication semantics (by removing the buffers), then the process would *deadlock* after a single reduction, because both the child and the parent would be trying to perform output. Under fully asynchronous semantics (by converting every output prefixed process $\alpha.P$ into $(\alpha \mid P)$), there is a possibility for a *race-condition*, because the parent's greeting could arrive before his channel. This would result in a *type mismatch*, since the child would attempt to receive on the greeting, instead of the parents address.

Nevertheless, we can still write many programs in TinyPi which exhibit undesirable behaviour. For example, the very simple program

```
prog = p <- spawn (x >- x ? \y -> END)
```

which translates to

$$[\![\texttt{prog}]\!] = (\nu p)p(y)$$

can never perform any transitions and therefore should probably be rejected as invalid by a decent compiler.

To eliminate such problems, we will equip $\pi_{\texttt{dist}}$ with a type system, and as valid TinyPi programs consider only those programs which translate into well-typed $\pi_{\texttt{dist}}$ terms.

3.2 Typing $\pi_{\texttt{dist}}$

Type systems are one of the standard means of ensuring safety of programming languages. They are a particularly appealing tool because of their natural connection to logic, known as the Curry–de Bruijn–Howard correspondence.

In the world of sequential computations, which is formally captured by the λ-calculus, the concept of typability is very well understood. Naively, types can be interpreted as sets and programs as mappings between them. However, more interestingly, an intuitionistic reading of types allows to interpret them as propositions in intuitionistic logic. Well-typed programs then become witnesses, or *constructive* proofs, of these propositions.

When we equip a programming language with a type system, we have to show that it is sound with respect to the reduction semantics of programs. This is usually split up into two properties: progress and preservation. Preservation ensures that the typing of programs is invariant under reduction, while progress ensures that well-typed programs never get 'stuck' or exhibit undefined behaviour. In the presence of these two properties, we say that the type system guarantees *type-safety* of programs.

However, what does it mean to type a process calculus? In particular, how do we assign types to process terms, what is their denotational meaning and how do we interpret type-safety for (a programming language based on) a process calculus?

It is clear that the previous models of type theories (of sequential computation) no longer work. In particular, given a π-calculus process, it does not seem reasonable to interpret it as a mapping between two types or sets. In fact, it is fair to say that at the current moment there is no universally agreed interpretation of typability in the setting of process calculi.

However, numerous different type systems have been proposed and at least two major 'themes' can be identified.

The first one can be traced back to the earliest developments of the π-calculus (for an overview, see [KPT99]) and is what we shall coin the *linear* theme. In this theme, various linear type systems were proposed, where the word 'linear' is a reference to Girard's Linear Logic [Gir87]. These type systems aim to express the usage of channels by treating them as 'limited resources'. A reference to a linear channel usually means that a process must use the channel name for a *single* input or output operation.

The second and quite recent theme is of the so-called Session Type systems. Many structurally quite different typing disciplines are referred to as being Session Type systems, however a common feature is that they split up communication between processes into units of interaction called *sessions*. Sessions are expressed as types which are assigned to channels and denote the flow of information happening through them.

We will consider a particular incarnation of the Session Typing discipline developed by Toninho, Caires, and Pfenning, which can be seen as a successful combination of the two themes above.

3.2.1 A Session Type system: πMILL

The spread of distributed systems has also increased the need for a more formal treatment of communication and therefore Session Types have been a very active research topic in recent years ([Bet+08], [Pad12], [CP10], [GH05]).

One outcome of this research direction is the discovery of another beautiful connection between logic and type theory in the style of Curry–de Bruijn–Howard. In particular, [CPT13] and [Wad12] show that there is a correspondence between propositions in Girard's Linear Logic and (binary) Session Types. Toninho, Caires, and Pfenning formulate a correspondence between Dual Intuitionistic Linear Logic (DILL) and Session Types, while Wadler provides an alternative formulation connecting Classical Linear Logic and Session Types.

In the rest of this thesis, we will follow the former formulation of the correspondence. We choose this particular formulation over the one proposed by Wadler, because it has been developed much more broadly and also because we believe that an intuitionistic treatment reflects the computational 'content' of the correspondence in a better way.

The correspondence is displayed via a type system for the synchronous π-calculus with binary choice, called πDILL. The most pleasing property of πDILL is that it provides full type-safety in the style of Milner. Subject reduction (or preservation) in this case implies *session fidelity*: processes follow the communication protocol described by their type. Similarly, progress implies deadlock-freedom. Combined with linearity, the safety conditions also ensure *global liveness* or progress, which is not guaranteed by most of the other session type systems that only guarantee liveness within a single session.

To simplify the presentation, we will consider a reduced version of the type system πDILL, which we shall coin πMILL. πMILL will correspond to the fragment of linear logic known as Multiplicative Intuitionistic Linear Logic (MILL). We obtain πMILL by removing replication and choice from the π-calculus (which corresponds to removing exponentials and the additive connectives from the linear logic). This also means we no longer need to keep track of two contexts in the type judgements.

Figure 8 contains type judgement rules of πMILL. We note that since this is a linear type system, weakening and contraction are not permitted, otherwise the

Figure 8 The type system πMILL

$$\frac{\Delta \vdash P :: T}{\Delta, y : \mathbf{1} \vdash P :: T}\ (\mathbf{1}\mathsf{L}) \qquad \frac{}{\vdash \mathsf{end} :: x : \mathbf{1}}\ (\mathbf{1}\mathsf{R})$$

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

$$\frac{\Delta \vdash P :: y : A \qquad \Delta', x : B \vdash Q :: T}{\Delta, \Delta', x : A \multimap B \vdash (\nu y)\overline{x}\langle y\rangle.(P \mid Q) :: T}\ (\multimap\mathsf{L})$$

$$\frac{\Delta, y : A \vdash P :: x : B}{\Delta \vdash x(y).P :: x : A \multimap B}\ (\multimap\mathsf{R}) \qquad \frac{\Delta, y : A, x : B \vdash P :: T}{\Delta, x : A \otimes B \vdash x(y).P :: T}\ (\otimes\mathsf{L})$$

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

treatment is standard. The general form of the type judgements is

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \vdash P :: y : B$$

which says that the process P can offer the service (of type) B over the channel y, by using the services $A_1, \ldots, A_n$ offered over the channels $x_1, \ldots, x_n$ respectively. Since this is a linear type system, it is implied that the process must *fully* use the services. The right rules correspond to process formation, while the left rules describe service usage. Linear implication ($\multimap$) is used to denote *input*, while multiplicative conjunction ($\otimes$) is used to denote *output*. For example, the type

$$x : (\mathbf{1} \multimap \mathbf{1}) \otimes \mathbf{1}$$

says that the channel x offers a single output capability of another channel of type $\mathbf{1} \multimap \mathbf{1}$, which offers an input of a channel of unit type (i.e. a channel which offers no capabilities). We note that $\mathbf{1}$ is the only base type and therefore we consider only the communication of channel names. However, it is trivial to extend the type system with other kinds of 'persistent' primitive types, such as booleans or integers by interpreting them as exponential types.

Cut reduction (rule Cut) is interpreted as *communication*: it says that a process P offering a service A over a channel x, can be composed with another process *using* that service to provide a new service. Observe that the process contexts are *disjoint*, which ensures that there are no causal dependencies between the resources used by two processes.

Finally, observe that by erasing the process terms ('P ::') and channel names ('x :') from the typing judgements, we would get the usual formulation of MILL.

3.2.2 Recycling: safety (almost) for free

To keep things simple and aligned with the presented $\pi\mathsf{MILL}$ type system, we will consider a reduced version of $\pi_{\mathtt{dist}}$ with input-guarded choice removed and replaced by regular input prefixing, which we will call $\pi^{-}_{\mathtt{dist}}$.

It would be preferable to not have to reinvent the wheel and just 'directly' apply the type system $\pi\mathsf{MILL}$ (or in fact, $\pi\mathsf{DILL}$) to $\pi^{-}_{\mathtt{dist}}$. However, this is not possible because $\pi\mathsf{MILL}$ types *synchronous* terms, while our calculus is asynchronous (more precisely, FIFO-ordered).

In the paper titled "Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication" ([DeY+12]), a very interesting approach to typing communication over FIFO-buffered channels is proposed. First of all, an alternative – *asynchronous* – process term assignment to the types of $\pi\mathsf{DILL}$ is described, which allows to type processes in the (dyadic) asynchronous π-calculus.[9] (We will denote the type system obtained by this assignment as $\pi\mathsf{DILL}_a$.) Afterwards, the encoding from Example 2.1.1 is used to type communication under FIFO-buffered semantics, by encoding FIFO-buffered channels into π_a and then applying the type system $\pi\mathsf{DILL}_a$.

Following this approach would indeed provide us with a type-system for $\pi^{-}_{\mathtt{dist}}$ almost for 'free'. However, this approach is slightly unsatisfactory due to multiple reasons.

First of all, since the encoding in Example 2.1.1 is not compositional, the relationship between the source terms and the encoded terms can quickly become quite confusing. Moreover, a clear connection between the transition graph of the original and encoded processes is lost due to the relabeling of channels.

Secondly, this approach makes it impossible, or at least very inconvenient, to consider extensions to the type system. In particular, since the encoding forces process to only exhibit binary communication, we cannot expect to extend the type system to allow for more complex communication patterns.

However, a careful analysis of the derivation of the system $\pi\mathsf{DILL}_a$ in [DeY+12] reveals another possibility. While attempting to retrofit the type system $\pi\mathsf{DILL}$ with an asynchronous process assignment, the authors of [DeY+12] face two problems when they try to naively switch from synchronous to asynchronous output.

The first (obvious) problem is that two previously sequential outputs can now be received out of order, therefore violating, for example, the type assignment $A \multimap B \multimap C$, which implies that the name of type B should be input *after* the name of type A.

The second problem is that a process might now accidentally receive its own output. In particular, consider the well-typed process (here we extend the message

[9] Since π_a is a sub-calculus of π_{nc}, in theory $\pi\mathsf{DILL}$ could already implicitly be used to type asynchronous communication, however the synchronous process assignment relies on the presence of output-prefixing and therefore the class of well-typed asynchronous protocols would be very small.

types with primitive integers and booleans for clarity):

$$(\nu c)(P \mid Q) = (\nu c)(\overline{c}\langle 42\rangle.c(y) \mid c(x).\overline{c}\langle \mathsf{True}\rangle)$$

and its naive asynchronous translation

$$(\nu c)(\overline{c}\langle 42\rangle \mid c(y) \mid c(x).\overline{c}\langle \mathsf{True}\rangle)$$

Its clear that in the first case, the process always safely reduces to end, however in the translated version its possible that the process P now performs self-communication and receives the value 42 that it itself sent, violating the protocol and leaving the process Q stuck.

Not at all by accident, both of the situations can never arise in $\pi_{\mathtt{dist}}$. In particular, the first problem is alleviated because channels in $\pi_{\mathtt{dist}}$ are FIFO-buffered, while the second problem cannot occur at all, because of the *full ownership* restriction. In particular, is it not possible that there are two parallel processes which both use a single channel for sending and receiving. This gives us a hint for another adaptation of $\pi\mathsf{DILL}$.

As mentioned previously, the 'user' syntax of $\pi_{\mathtt{dist}}$ is identical to the syntax of the synchronous calculus π_{ic}, except for the additional syntactic restrictions. This means that, in some cases, we can 'pretend' that $\pi_{\mathtt{dist}}$ processes communicate using synchronous channels. In particular, we will denote $S\pi_{\mathtt{dist}}$ to be the process calculus obtained by applying the syntactic restrictions of $\pi_{\mathtt{dist}}$ to the calculus π_{ic}. This allows for an alternative – synchronous – reading of $\pi_{\mathtt{dist}}$. Analogously, we define $S\pi^{-}_{\mathtt{dist}}$ as a synchronous interpretation of $\pi^{-}_{\mathtt{dist}}$.

We are interested in such a reading because, for certain kinds of protocols, we are morally allowed to ignore the existence of channel buffers and consider process reductions 'modulo buffer actions'. This is the case when the presence of intermediate buffers does not impact the flow of information or the fully-reduced states of processes. In fact, our next claim is that, under the additional syntactic restrictions of $\pi_{\mathtt{dist}}$, well-typed $\pi\mathsf{MILL}$ processes communicate using precisely such protocols. This means we can *pretend* that the user syntax of $\pi^{-}_{\mathtt{dist}}$ represents a *synchronous* process calculus and type-check its processes using $\pi\mathsf{MILL}$, but execute the well-typed processes using FIFO-buffered communication semantics! This might sound a bit confusing, but will hopefully become more clear along the way as we state and prove this fact formally.

Before continuing further, we summarise our inventory of π-calculi:

- $S\pi_{\mathtt{dist}}$ is π_{ic} with the additional syntactic restrictions of $\pi_{\mathtt{dist}}$, i.e. 'synchronous' $\pi_{\mathtt{dist}}$.
- $\pi^{-}_{\mathtt{dist}}$ is $\pi_{\mathtt{dist}}$ without choice.
- $S\pi^{-}_{\mathtt{dist}}$ is $S\pi_{\mathtt{dist}}$ without choice, i.e. 'synchronous' $\pi^{-}_{\mathtt{dist}}$, i.e. π_{nc} with the additional syntactic restrictions of $\pi_{\mathtt{dist}}$.
- $\pi\mathsf{MILL}$, when referred to as process calculus, is the sub-calculus of well-typed processes of π_{nc}.
- $S\pi^{-}_{\mathtt{dist}}\mathsf{MILL}$ is the sub-calculus of $S\pi^{-}_{\mathtt{dist}}$ of well-typed processes.
- $\pi^{-}_{\mathtt{dist}}\mathsf{MILL}$ is the sub-calculus of $\pi^{-}_{\mathtt{dist}}$ of well-typed processes.

When we mention well-typed processes we assume that they are typed using *some* type system, which we will define later.

At this point we only know that $\pi\mathsf{MILL}$ ensures type-safety of π_{nc} processes. Next, we will show that a small modification of $\pi\mathsf{MILL}$ also ensures type-safety for the calculus $S\pi^{-}_{\mathtt{dist}}\mathsf{MILL}$. Finally, with a little trick and some work we will show how to type-check $\pi^{-}_{\mathtt{dist}}$ processes.

3.2.3 $\pi\mathsf{MILL}$ Processes Need Not (Always) Be Synchronous

In this section, our goal is to prove the main result of this thesis.

We wish to show that, by applying the additional restrictions of $\pi_{\mathtt{dist}}$ to the process calculus of (well-typed) $\pi\mathsf{MILL}$ processes, we obtain a sub-calculus with processes which can be *safely* executed under both synchronous and FIFO-buffered communication semantics.

The particularly appealing outcome of this fact is that processes can execute communication protocols with a much greater degree of independence, which both improves the performance of such protocols and allows to execute them in distributed settings.

We first of all have to determine whether $\pi\mathsf{MILL}$ provides type-safety for $S\pi^{-}_{\mathtt{dist}}$ processes.

We do this in a slightly hand-wavy manner, since it is more or less straightforward. In particular, since the reduction semantics of $S\pi^{-}_{\mathtt{dist}}$ is the same as π_{nc}, we only need to review what effect the $\pi_{\mathtt{dist}}$ restrictions have on the process assignments. In particular, the only problem that arises is that we can get well-typed terms which cannot be composed with their *parallel representatives* ([Pér+12, Def. 6.5]). This does not mean that such terms exhibit unsafe behaviour, but rather that processes which are *dual* to them, produce invalid $S\pi^{-}_{\mathtt{dist}}$ terms when composed with them.

As an example, consider the following well-typed $\pi\mathsf{MILL}$ process P:

$$c : \mathbf{1} \multimap \mathbf{1} \otimes \mathbf{1} \vdash (\nu x)\overline{c}\langle x\rangle.c(y) :: x : \mathbf{1} \tag{3.2.1}$$

Note that P is also a valid $S\pi^{-}_{\mathtt{dist}}$ term. In order to compose this process, we need to have another well-typed process Q, which provides the service along channel c, which is used by P. The type of c implies that Q would have to perform both input and output on the channel c, so if we applied Cut along the channel c, we would obtain the process $(\nu c)(P \mid Q)$, which is an *invalid* $S\pi^{-}_{\mathtt{dist}}$ term, because it violates full ownership: both P and Q perform input on the name c.

We could turn a blind eye on this issue via some appeal to 'syntactical-well-formedness', but we might as well just figure out how to fix it. We recursively

define the notation of iterated operations:

$$\multimap_i^0 A_i := \mathbf{1}$$
$$\multimap_i^{(n+1)} A_i := A_{n+1} \multimap (\multimap_i^n A_i)$$

$$\otimes_i^0 B_i := \mathbf{1}$$
$$\otimes_i^{(n+1)} B_i := B_{n+1} \otimes (\otimes_i^n B_i)$$

for some sequences of A_i's and B_i's. We also omit the superscript n when it is irrelevant or inferable.

We now have the following proposition.

Proposition 3.2.1. *Ensuring that well-typed π_{nc} processes are also (syntactically) well-formed $S\pi^-_{\mathtt{dist}}$ processes is equivalent to restricting all the type formation rules in Figure 8 in a way that ensures that all types have one of the two forms: $\multimap_i A_i$ or $\otimes_i B_i$.*

Proof. (Note: the restriction mentioned in the proposition can be implemented as a simple side-condition in each rule of Figure 8. For example, in the rule $\multimap$R, since the channel x is being assigned the type $x : A \multimap B$, we require that B also has the shape $B = \multimap_i B_i$. Similarly for other rules. Alternatively, separate type formation rules can be presented.)

There are two syntactic conditions we need to consider: the localised channels restriction (LCr) and the full-ownership restriction (FOr). The LCr is relevant whenever new input prefixes are formed, while the FOr is relevant whenever a parallel composition appears.

We consider only a couple of rules from Figure 8 as an example:

- Rule $\multimap$R: from the localised channels restriction (LCR), we infer that y cannot be used for input and therefore its type A must be of the form $\multimap_i A_i$.
- Rule $\otimes$L: similarly, from the LCr we infer that y is typed as input-only (i.e. $\multimap_i A_i$) and, further, that B must be of the form $\otimes_i B_i$ (since otherwise it is possible to apply cut on x and violate FOr).

Similarly for the other rules. □

We therefore define the type system $S\pi^-_{\mathtt{dist}}$MILL by the rules in Figure 8 with the additional restriction from Proposition 3.2.1. We can then finally claim the following:

Proposition 3.2.2. *$S\pi^-_{\mathtt{dist}}$MILL ensures type-safety of $S\pi^-_{\mathtt{dist}}$ processes, i.e. $S\pi^-_{\mathtt{dist}}$MILL has subject reduction and progress.*

Proof. From the fact that πMILL provides type-safety to π_{nc} processes and Proposition 3.2.1. □

We now move on to our main goal: showing that the type-safety of $S\pi^-_{\mathtt{dist}}\mathsf{MILL}$ is preserved when we move from synchronous to FIFO-buffered communication. In particular, this will allow us to type-check $\pi^-_{\mathtt{dist}}$ processes (that are expressed using only the 'user' syntax) using the type rules of $S\pi^-_{\mathtt{dist}}\mathsf{MILL}$ and ensure safety of their execution.

We first of all note that the restriction from Proposition 3.2.1 is even more crucial for $\pi^-_{\mathtt{dist}}$ than it was for $S\pi^-_{\mathtt{dist}}$. Consider again the process P in 3.2.1. Under FIFO-buffered channel semantics, this process no longer merely lacks a parallel representative, but actually violates the intended session type protocol. In particular, if we allowed P to be well-typed, it could output the name x and later receive it back itself in the next reduction. However, the restrictions from Proposition 3.2.1 ensure that all channels are used exclusively for either input or output, therefore such situations cannot occur. In particular, P could not be well-typed.

In order to formally relate synchronous and FIFO-buffered communication semantics of $S\pi^-_{\mathtt{dist}}\mathsf{MILL}$ processes, we define a translation function from $S\pi^-_{\mathtt{dist}}\mathsf{MILL}$ to $\pi^-_{\mathtt{dist}}$.

Definition 3.2.1 (Encoding into FIFO-buffered semantics)**.** We define the translation function

$$(\!|\cdot|\!)\colon S\pi^-_{\mathtt{dist}}\mathsf{MILL} \to \pi^-_{\mathtt{dist}}$$

simply as the identity embedding between the two syntaxes.

The co-domain of $(\!|\cdot|\!)$ is a subset of processes expressed in the 'user' syntax part of $\pi_{\mathtt{dist}}$.

We can now define formally what it means for a $\pi^-_{\mathtt{dist}}$ process to be well-typed:

Definition 3.2.2 (Well-typed $\pi^-_{\mathtt{dist}}$ process)**.** We say that $P \in \pi^-_{\mathtt{dist}}$ is *well-typed* if there exists $Q \in S\pi^-_{\mathtt{dist}}\mathsf{MILL}$, such that

$$(\!|Q|\!) = P$$

where $\pi^-_{\mathtt{dist}}$ and $(\!|\cdot|\!)$ is the translation from Definition 3.2.1.

We denote the sub-calculus of well-typed $\pi^-_{\mathtt{dist}}$ processes as $\pi^-_{\mathtt{dist}}\mathsf{MILL}$, and write $P \in \pi^-_{\mathtt{dist}}\mathsf{MILL}$ to mean that P is a well-typed $\pi^-_{\mathtt{dist}}$ process according to the above definition.

When we talk about type judgements $\Delta \vdash P :: x : A$ for some $P \in \pi^-_{\mathtt{dist}}\mathsf{MILL}$, we mean that there exists a $Q \in S\pi^-_{\mathtt{dist}}\mathsf{MILL}$, such that $(\!|Q|\!) = P$ and $\Delta \vdash Q :: x : A$.

Observe that since $(\!|\cdot|\!)$ is the identity embedding, $(\!|Q|\!) = P$ is equivalent to saying that $P = Q'$, for some $Q' \in S\pi^-_{\mathtt{dist}}$.

In order to show that this definition of well-typedness ensures safety, we show that there is an operational correspondence between $S\pi^-_{\mathtt{dist}}\mathsf{MILL}$ and $\pi^-_{\mathtt{dist}}\mathsf{MILL}$ processes via a series of propositions.

We start with the simplest proposition, which says that the translated terms can always mimic the reductions of the source terms.

Proposition 3.2.3 (Operational completeness of $(\!|\cdot|\!)$)**.** *For all* $P, P' \in S\pi^{-}_{\mathtt{dist}}\mathsf{MILL}$*,*

$$P \longrightarrow P' \text{ implies } (\!|P|\!) \Longrightarrow (\!|P'|\!)$$

Proof. Since P is well-typed, we have that

$$\Delta \vdash P :: z : T$$

for some context Δ, channel z and type T.

We prove the lemma by induction on the derivation of the processes type judgement. The proof is essentially the same as the proof of subject reduction for $\pi\mathsf{DILL}$ ([CPT13, Thm. A.1]), we do inversion on the type judgements to consider the possible cases.

In particular, by case analysis on the type judgments, we get that $P \longrightarrow P'$ implies that the last application was either Cut or $\mathbf{1}\mathsf{L}$, because all the other rules produce prefixed terms.

Case Cut By inversion, we can infer that:

- $P \equiv (\nu x)(P_1 \mid P_2)$
- $\Delta \equiv \Delta_1, \Delta_2$
- $\Delta_1 \vdash P_1 :: x : A$
- $\Delta_2, x : A \vdash P_2 :: z : T$

By case analysis on the LTS and type judgements, we get that only the following sub-cases are possible:

$P_1 \longrightarrow P_1'$ Since P_1 is structurally smaller than P, we are done by the induction hypothesis.

$P_2 \longrightarrow P_2'$ Same as the previous case.

$P_1 \xrightarrow{(\nu y)\overline{x}\langle y\rangle} P_1'$ and $P_2 \xrightarrow{x(y)} P_2'$

Appealing to Lemma A.7 in [CPT13], we have that it's sufficient to consider:

$$\begin{aligned} P_1 &\equiv (\nu y)\overline{x}\langle y\rangle.P_1' \\ P_2 &\equiv x(z).P_2' \\ P \longrightarrow P' &\equiv (\nu x)(\nu y)(P_1' \mid P_2\{y/z\}) \end{aligned}$$

Therefore, we can infer that

$$\begin{array}{rcl} (\!|P|\!) & \equiv & (\nu x)((\nu y)\overline{x}\langle y\rangle.P_1' \mid x(z).P_2' \mid x{:}[]) \\ & \longrightarrow & (\nu x)((\nu y)(P_1' \mid x{:}[y]) \mid x(z).P_2') \\ & \longrightarrow & (\nu x)(\nu y)(P_1' \mid P_2'\{y/z\} \mid x{:}[]) \\ & \equiv & (\!|P'|\!) \end{array}$$

as desired.

$P_1 \xrightarrow{x(y)} P_1'$ and $P_2 \xrightarrow{(\nu y)\overline{x}\langle y\rangle} P_2'$ Similar to the previous case.

Case 1L Since this does not affect the terms structure, it follows from the IH.

□

Observe that in the previous lemma we actually showed an even stronger result, namely

$$P \longrightarrow P' \text{ implies } [\![P]\!] \longrightarrow\!\!\!\!\longrightarrow [\![P']\!]$$

This is because we can always simulate a synchronous reduction, by two reductions in $\pi_{\mathtt{dist}}$: one that enqueues the message and another one that dequeues. In fact, we could have even avoided proving the previous proposition explicitly, by just referring to this observation.

However, by inspecting the previous proof we can see that a non-reducible $S\pi^{-}_{\mathtt{dist}}\mathsf{MILL}$ term cannot become reducible after a translation. This is captured in the following lemma.

Lemma 3.2.1. *For all $P \in S\pi^{-}_{\mathtt{dist}}$MILL,*

$$P \longrightarrow \text{ iff } (\!|P|\!) \longrightarrow$$

Proof. The *only if* part follows from Proposition 3.2.3.

The *if* part follows by a very similar case analysis. Namely, $(\!|P|\!)$ also reduces only if it is unprefixed. However, this means that P's derivation was exactly the same as the cases analysed in Proposition 3.2.3 and therefore P also reduces. □

Next, we note that we obviously do not have that $(\!|P|\!) \longrightarrow Q$ implies that $Q \equiv (\!|P'|\!)$, because, as we already observed, for every reduction in $S\pi^{-}_{\mathtt{dist}}$, two reductions in $\pi^{-}_{\mathtt{dist}}$ are needed. More generally, the translated terms do not necessarily follow the same reduction path as the original synchronous terms, i.e.

$$(\!|P|\!) \longrightarrow\!\!\!\!\longrightarrow Q \text{ does } \textit{not} \text{ imply } Q \equiv (\!|P'|\!)$$

As a counter-example, consider the following derivation:

Example 3.2.1 (Asynchronous reduction)**.** Consider the process $PQ \in S\pi^{-}_{\mathtt{dist}}\mathsf{MILL}$, defined by:

$$PQ = (\nu x)(P \mid Q) = (\nu x)((\nu y_1)\overline{x}\langle y_1\rangle.(\nu y_2)\overline{x}\langle y_2\rangle \mid x(z_1).x(z_2))$$

We then have that

$$\begin{array}{rcl}
(\!|(\nu x)(P \mid Q)|\!) & \equiv & (\nu x)((\!|P|\!) \mid (\!|Q|\!) \mid x{:}[]) \\
& \longrightarrow\equiv & (\nu x)((\nu y_2)\overline{x}\langle y_2\rangle \mid (\nu y_1)(x{:}[y_1]) \mid (\!|Q|\!)) \\
& \longrightarrow\equiv & (\nu x)((\nu y_1)(\nu y_2)(x{:}[y_1, y_2]) \mid (\!|Q|\!)) \\
& \not\equiv & (\!|P'|\!), \text{ for any } P', \text{ nevertheless:} \\
& \longrightarrow\equiv & (\nu x)((\nu y_2)(x{:}[y_2]) \mid x(z_2)) \\
& \longrightarrow\equiv & \mathsf{end} = (\!|\mathsf{end}|\!)
\end{array}$$

However, while the translated term did not follow the same reduction path as original, *eventually* it reduced to end, which is reachable from PQ.

Our next claim is that this will always happen, i.e. that the reductions of $\pi^{-}_{\mathtt{dist}}\mathsf{MILL}$ are confluent and can always reach a 'well-typed state'.

Proposition 3.2.4 (Confluence)**.** *For all $P \in S\pi^{-}_{\mathtt{dist}}\mathit{MILL}$, for all $Q_1, Q_2 \in \pi_{\mathtt{dist}}$, whenever*

$$(\!|P|\!) \Longrightarrow Q_1 \wedge (\!|P|\!) \Longrightarrow Q_2$$

then there exists a process P', such that

$$Q_1 \Longrightarrow (\!|P'|\!) \wedge Q_2 \Longrightarrow (\!|P'|\!)$$

holds.

Proof. Sketch: The proof is once again by induction on the derivation of the type judgement of P. Assume the premise of the proposition's implication holds. By Lemma 3.2.1, we have that $(\!|P|\!) \longrightarrow$ implies $P \longrightarrow$. From this, similarly as in the proof of Proposition 3.2.3, we can again conclude that the last rule applied was either Cut or $\mathbf{1}\mathsf{L}$.

The main case we have to consider is once again Cut. By inversion we have that

$$(\!|P|\!) \equiv (\nu x)((\nu y)\overline{x}\langle y\rangle.P_1 \mid x(z).P_2 \mid x{:}[])$$

and

$$(\!|P|\!) \longrightarrow (\nu x)((\nu y)(P_1 \mid x{:}[y]) \mid x(z).P_2)$$

Now, by the full-ownership restriction, we have that P_2 owns x, therefore P_1 can only append to the buffer of x. We can then perform sub-induction on (some well-ordering of) the reduction paths of P_1 and P_2. In particular, from the fact that $S\pi^{-}_{\mathtt{dist}}\mathsf{MILL}$ processes are confluent [CPT13, Thm. 5.2], we can choose a 'convenient' reduction paths of P_1 and P_2, where they *maximally* delay further communication with each other. We will then get a smallest possible term among the reduction paths that do not touch the buffer again, and we can then perform a single reduction that returns us to a well-typed $S\pi^{-}_{\mathtt{dist}}$ term, on which we can then apply the IH. □

From this, we can immediatelly conclude the following:

Corollary 3.2.1 (Operational soundness)**.** *For all $P, P' \in S\pi^{-}_{\mathtt{dist}}\mathit{MILL}$,*

$$(\!|P|\!) \Longrightarrow (\!|P'|\!) \text{ implies } P \Longrightarrow P'$$

We now finally arrive at an important conclusion.

Theorem 3.2.1 (Operational correspondance)**.** *For all P, P' in $S\pi^{-}_{\mathtt{dist}}\mathit{MILL}$,*

$$P \Longrightarrow P' \text{ iff } (\!|P|\!) \Longrightarrow (\!|P'|\!)$$

Proof. By Proposition 3.2.3 and Corollary 3.2.1. □

Using the above theorem we can conclude preservation and progress.

Corollary 3.2.2 ((Weak) Subject reduction)**.** *Let* $P, P' \in \pi^{-}_{\mathtt{dist}}\mathsf{MILL}$ *and assume that*

$$\Delta \vdash P :: x : T$$

then $P \Longrightarrow P'$ *implies that*

$$\Delta \vdash P' :: x : T$$

.

Proof. P, $P' \in \pi^{-}_{\mathtt{dist}}\mathsf{MILL}$ implies that P, $P' \equiv (\!|R|\!)$, $(\!|R'|\!)$ for some R, $R' \in S\pi^{-}_{\mathtt{dist}}\mathsf{MILL}$. By Theorem 3.2.1 we have that $R \Longrightarrow R'$ and the result follows from the fact that $S\pi^{-}_{\mathtt{dist}}\mathsf{MILL}$ has subject reduction. □

To state progress we have to define when a process has a 'potential' for reduction:

Definition 3.2.3 (Live process)**.** We say that a process π-calculus process P is *live*, if $P \equiv (\nu \vec{n})(\alpha.P' \mid R)$, for some vector of names $\vec{n}$, prefix α and processes P', R.

We can now deduce progress. We use the notation $\cdot \vdash P :: x : A$ to mean that P is typed under the empty context.

Corollary 3.2.3 (Progress)**.** *If* $P \in \pi^{-}_{\mathtt{dist}}\mathsf{MILL}$, $\cdot \vdash P :: z : \mathbf{1}$ *and* P *is* live, *then* $P \Longrightarrow Q$ *for some* $Q \in \pi^{-}_{\mathtt{dist}}\mathsf{MILL}$.

Proof. We have that $P \equiv (\!|R|\!)$, for some R. Since $(\!|\cdot|\!)$ is the identity embedding, R is also live and $\cdot \vdash R :: z : \mathbf{1}$, hence by progress of $S\pi^{-}_{\mathtt{dist}}\mathsf{MILL}$ R reduces to some R'. Finally, by Theorem 3.2.1 we have that $P \equiv (\!|R|\!) \Longrightarrow (\!|R'|\!) \equiv P'$, for some $P' \in \pi^{-}_{\mathtt{dist}}\mathsf{MILL}$, as required. □

4 Outro

We now sum up our journey towards taming π-calculus and Session Types for distributed programming languages.

4.1 Summary of the results

Arguably, for the biggest part, this thesis is concerned with motivating and then developing the $\pi_{\mathtt{dist}}$ calculus. Why would someone develop yet another flavour of π-calculus, when there are already too many of them?

One of the main inspirations for doing this, in particular, for attempting to develop a π-calculus with FIFO-buffered channels, was the realisation that Session Types are about *ordered* communication. More precisely, the type of the channel expresses the flow of information through it, which is ordered. Considering a formalism (or a runtime that implements it), which automatically preserves the order of information flow within units of interaction, allows to

remove some strain on the possible communication patterns, as discussed in subsubsection 3.1.2.

The variety of encodings and embeddings appearing in the previous section might cause confusion, so we summarise the story in a slightly simplified way.

Our goal was to ensure type-safety of TinyPi programs. TinyPi programs are translated (or *compiled*) to $\pi_{\texttt{dist}}$ processes, so we reduced our goal to ensuring type-safety of $\pi_{\texttt{dist}}$ processes. To do that, we took the 'user' syntax of $\pi_{\texttt{dist}}$, removed choice from it and pretended this was a synchronous calculus. We could then, with a little bit of tweaking, apply the type system $\pi\mathsf{MILL}$ to type $\pi_{\texttt{dist}}$ processes. We were *allowed* do this, because we proved that the 'original' type-safety of $\pi\mathsf{MILL}$ is preserved under FIFO-buffered communication semantics (and a few extra constraints).

The important point here is that in both steps we were working with the *user* syntax of $\pi_{\texttt{dist}}$, therefore a programmer working with TinyPi does not need to *see* the FIFO buffers. The channel buffers become an implementation detail, which is not visible in the high-level syntax.

Unfortunately, we showed our results only for $\pi^{-}_{\texttt{dist}}$, which does not contain choice. Does the preservation of type-safety carry over if we extend $\pi\mathsf{MILL}$ with additive connectives (i.e. obtain correspondence with the 'full' intuitionistic linear logic) and restore choice in $\pi_{\texttt{dist}}$? We do not have a proof, but we believe that the answer essentially depends on what degree of expressiveness is permitted for the (typed) choice operator.

In particular, if we are only interested in so-called *labelled* choice, which represents standard sum types and appears in the process assignment for $\pi\mathsf{DILL}$, then we think the answer is "yes". In particular, the extra branching introduced by this flavour of choice should not cause any problems, because the linear typing discipline ensures that no interference is possible on a global level, therefore the overall situation does not change even under FIFO-buffered communication.

However, if we wanted to somehow type the full-power of input-guarded choice – namely, the fact that it can operate on *channels* – then we believe the answer would be negative, because $\pi\mathsf{DILL}$ is not powerful enough to handle such cases. In particular, this would imply that the choice branches are dependent on input from *multiple* channels, which exits the realm of binary protocols that $\pi\mathsf{DILL}$ types.

Finally, we would like to go back to Example 3.2.1 for a moment to appreciate the practical outcomes of the result.

FIFO-buffered (and more generally, asynchronous) communication semantics is very appealing in implementations, because it allows communicating processes to execute independently, which greatly improves *parallelism* of the programs. In particular, as Example 3.2.1 shows, the process PQ has the freedom to execute via multiple paths, however since it is well-typed, none of the paths can lead to something 'going wrong'.

The benefits of asynchronous protocols have always been obvious in distributed settings, e.g. where two computers communicate over the network and network

latency is the bottleneck of the distributed system. On the other hand, it is sometimes argued that for tightly coupled local computations, synchronous concurrency is preferred because it avoids the use of any buffers, and therefore can be more efficient both in space and time. Nevertheless, modern computing device architectures are increasingly changing in a way which makes them, even locally, resemble a distributed system, where communication latency is, once again, the main bottleneck of the system. This makes asynchronous communication appealing even in local settings.

Another argument against asynchronous protocols is that it is much harder to reason about correctness of both asynchronous protocols and the programs implementing them. However, by equipping our calculus with a FIFO-buffered session type system, we are able to enjoy the benefits of decoupled asynchronous execution while retaining full safety.

4.2 Future work and conclusion

As the title of the thesis suggests, we, unfortunately, feel there are more things that we did *not* manage to do, rather than that we did.

In particular, although we managed to describe $\pi_{\texttt{dist}}$, it can be argued that this description is not convenient to work with. Therefore, in future work, at the very least it would be interesting to develop a suitable notion of behavioural equivalence for FIFO-ordered communication. Moreover, it is clear that the behavioural aspects of $\pi_{\texttt{dist}}$ are quite different from other π-calculi. In order to analyse this, we would want to lift all the syntactic constraints into a type system. It would then be possible to, for example, develop behavioural equivalences for typed-contexts, as it is done in, e.g. [CPT13].

Going even further with this, it is actually an interesting question whether $\pi_{\texttt{dist}}$ would benefit from a completely different formulation, which is not based on the π-calculus. In particular, we have seen that the full-ownership restriction does not mix well with the general formulation of the π-calculus.

We would also very much like to extend the type system of $\pi\mathsf{DILL}$ to be able to handle *symmetric* protocols, where two participants communicate by executing identical actions. These kind of protocols appear, when one considers problems such as leader-election, and are unfortunately untypeable in $\pi\mathsf{DILL}$, because it assumes binary, *dual* communication.

In summary, we have barely scratched the opportunity for a slightly different approach to the formulation of π-calculi, session typing and, in general, formal treatment of distributed communication, but we hope that this work might serve as an inspiration for future research in this direction.

References

[ACS96] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. “On Bisimulations for the Asynchronous pi-Calculus”. In: *Proceedings*

of the 7th International Conference on Concurrency Theory. CONCUR '96. London, UK, UK: Springer-Verlag, 1996, pp. 147–162. ISBN: 3-540-61604-7. URL: http://dl.acm.org/citation.cfm?id=646731.703844.

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986. ISBN: 0-262-01092-5.

[Ama00] Roberto M. Amadio. "On Modelling Mobility". In: *Theor. Comput. Sci.* 240.1 (June 2000), pp. 147–176. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(99)00230-3. URL: http://dx.doi.org/10.1016/S0304-3975(99)00230-3.

[Arm07] Joe Armstrong. "A History of Erlang". In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: ACM, 2007. ISBN: 978-1-59593-766-7. DOI: 10.1145/1238844.1238850. URL: http://doi.acm.org/10.1145/1238844.1238850.

[Bar85] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1985. ISBN: 9780080933757.

[Bau+09] Andrew Baumann et al. "The multikernel: a new OS architecture for scalable multicore systems". In: *SOSP*. 2009, pp. 29–44.

[Bet+08] Lorenzo Bettini et al. "Global Progress in Dynamically Interleaved Multiparty Sessions". In: *CONCUR*. 2008, pp. 418–433.

[BK84] Jan A. Bergstra and Jan Willem Klop. "Process Algebra for Synchronous Communication". In: *Information and Control* 60.1-3 (1984), pp. 109–137. DOI: 10.1016/S0019-9958(84)80025-X. URL: http://dx.doi.org/10.1016/S0019-9958(84)80025-X.

[Bou87] Luc Bougé. "Repeated Snapshots in Distributed Systems with Synchronous Communications and Their Implementation in CSP". In: *Theor. Comput. Sci.* 49.2-3 (Jan. 1987), pp. 145–169. ISSN: 0304-3975. DOI: 10.1016/0304-3975(87)90005-3. URL: http://dx.doi.org/10.1016/0304-3975(87)90005-3.

[Bou88] Luc Bougé. "On the Existence of Symmetric Algorithms to Find Leaders in Networks of Communicating Sequential Processes." In: *Acta Inf.* 1988, pp. 179–201.

[Bou92] Gérard Boudol. *Asynchrony and the Pi-calculus*. English. Research Report RR-1702. INRIA, 1992, p. 15. URL: http://hal.inria.fr/inria-00076939.

[BPV08] Romain Beauxis, Catuscia Palamidessi, and Frank D. Valencia. "On the Asynchronous Nature of the Asynchronous pi-Calculus". In: *Concurrency, Graphs and Models*. 2008, pp. 473–492.

[BZ83] Daniel Brand and Pitro Zafiropulo. "On Communicating Finite-State Machines". In: *J. ACM* 30.2 (1983), pp. 323–342. DOI: 10.1145/322374.322380. URL: http://doi.acm.org/10.1145/322374.322380.

[CMT96] Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. "Synchronous, Asynchronous, and Causally Ordered Communication". In: *Distributed Computing*. 1996, pp. 173–191.

[CP10] Luís Caires and Frank Pfenning. "Session Types as Intuitionistic Linear Propositions". In: *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*. 2010, pp. 222–236. DOI: 10.1007/978-3-642-15375-4_16. URL: http://dx.doi.org/10.1007/978-3-642-15375-4_16.

[CPT13] Luís Caires, Frank Pfenning, and Bernardo Toninho. "Linear Logic Propositions as Session Types". In: *Mathematical Structures in Computer Science* (2013). To appear. Special Issue on Behavioural Types. URL: http://www.cs.cmu.edu/~fp/papers/mscs13.pdf (visited on 12/10/2014).

[Den+13] Xiaojie Deng, Yu Zhang, Yuxin Deng, and Farong Zhong. "The Buffered π-Calculus: A Model for Concurrent Languages". In: *Language and Automata Theory and Applications - 7th International Conference, LATA 2013, Bilbao, Spain, April 2-5, 2013. Proceedings*. 2013, pp. 250–261. DOI: 10.1007/978-3-642-37064-9_23. URL: http://dx.doi.org/10.1007/978-3-642-37064-9_23.

[DeY+12] Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. "Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication". In: *CSL*. 2012, pp. 228–242.

[FG00] Cédric Fournet and Georges Gonthier. "The Join Calculus: A Language for Distributed Mobile Programming". In: *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*. 2000, pp. 268–332. DOI: 10.1007/3-540-45699-6_6. URL: http://dx.doi.org/10.1007/3-540-45699-6_6.

[FG05] Cédric Fournet and Georges Gonthier. "A hierarchy of equivalences for asynchronous calculi". In: *J. Log. Algebr. Program.* 63.1 (2005), pp. 131–173.

[GGS13] Rob J. van Glabbeek, Ursula Goltz, and Jens-Wolfhard Schicke-Uffmann. "On Characterising Distributability". In: *Logical Methods in Computer Science* 9.3 (2013). DOI: 10.2168/LMCS-9(3:17)2013. URL: http://dx.doi.org/10.2168/LMCS-9(3:17)2013.

[GH05] Simon J. Gay and Malcolm Hole. "Subtyping for session types in the pi calculus". In: *Acta Inf.* 42.2-3 (2005), pp. 191–225.

[Gir87] Jean-Yves Girard. "Linear logic". In: *Theoretical computer science* 50.1 (1987), pp. 1–101.

[Gla12] Rob J. van Glabbeek. "Musings on Encodings and Expressiveness". In: *Proceedings Combined 19th International Workshop on Expressiveness in Concurrency and 9th Workshop on Structured Operational Semantics, EXPRESS/SOS 2012, Newcastle upon Tyne, UK, September 3, 2012*. 2012, pp. 81–98. DOI: 10.4204/EPTCS.89.7. URL: http://dx.doi.org/10.4204/EPTCS.89.7.

[GN14] Daniele Gorla and Uwe Nestmann. "Full Abstraction for Expressiveness: History, Myths and Facts". In: *Mathematical Structures in Computer Science* (2014). To appear.

[Gor10] Daniele Gorla. "Towards a unified approach to encodability and separation results for process calculi". In: *Inf. Comput.* 208.9 (2010), pp. 1031–1053. DOI: 10.1016/j.ic.2010.05.002. URL: http://dx.doi.org/10.1016/j.ic.2010.05.002.

[GPT07] Robert Griesemer, Rob Pike, and Ken Thompson. *The Go Programming Language*. 2007. URL: https://golang.org/ (visited on 12/29/2014).

[Gra78] Jim Gray. "Notes on Data Base Operating Systems". In: *Operating Systems, An Advanced Course*. London, UK, UK: Springer-Verlag, 1978, pp. 393–481. ISBN: 3-540-08755-9. URL: http://dl.acm.org/citation.cfm?id=647433.723863.

[HM00] Joseph Y. Halpern and Yoram Moses. "Knowledge and common knowledge in a distributed environment". In: *CoRR* cs.DC/0006009 (2000). URL: http://arxiv.org/abs/cs.DC/0006009.

[Hoa78] C. A. R. Hoare. "Communicating Sequential Processes". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585. URL: http://doi.acm.org/10.1145/359576.359585.

[HT91] Kohei Honda and Mario Tokoro. "An Object Calculus for Asynchronous Communication". In: *ECOOP'91 European Conference on Object-Oriented Programming, Geneva, Switzerland, July 15-19, 1991, Proceedings*. 1991, pp. 133–147. DOI: 10.1007/BFb0057019. URL: http://dx.doi.org/10.1007/BFb0057019.

[HY95] Kohei Honda and Nobuko Yoshida. "On Reduction-Based Process Semantics". In: *Theor. Comput. Sci.* 151.2 (1995), pp. 437–486. DOI: 10.1016/0304-3975(95)00074-7. URL: http://dx.doi.org/10.1016/0304-3975(95)00074-7.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. "Multiparty asynchronous session types". In: *In Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*. ACM Press, 2008, pp. 273–284.

[Kah74] Gilles Kahn. "The Semantics of Simple Language for Parallel Programming". In: *IFIP Congress*. 1974, pp. 471–475.

[KPT99] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. "Linearity and the pi-calculus". In: *ACM Trans. Program. Lang. Syst.* 21.5 (1999), pp. 914–947.

[Lév97] Jean-Jacques Lévy. "Some results in the join-calculus". English. In: *Theoretical Aspects of Computer Software*. Ed. by Martín Abadi and Takayasu Ito. Vol. 1281. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997, pp. 233–249. ISBN: 978-3-540-63388-4. DOI: 10.1007/BFb0014554. URL: http://dx.doi.org/10.1007/BFb0014554.

[Maz86] Antoni W. Mazurkiewicz. “Trace Theory”. In: *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986*. 1986, pp. 279–324. DOI: 10.1007/3-540-17906-2_30. URL: http://dx.doi.org/10.1007/3-540-17906-2_30.

[Mer00] Massimo Merro. “Locality in the π-calculus and applications to distributed objects”. PhD thesis. Ecole des Mines, France, 2000.

[Mer99] Massimo Merro. “On Equators in Asynchronous Name-passing Calculi without Matching”. In: *Electr. Notes Theor. Comput. Sci.* 27 (1999), pp. 57–70. DOI: 10.1016/S1571-0661(05)80295-6. URL: http://dx.doi.org/10.1016/S1571-0661(05)80295-6.

[Mil80] Robin Milner. “A Calculus of Communicating Systems”. In: 1980.

[Mil93] Robin Milner. “Elements of Interaction: Turing Award Lecture”. In: *Commun. ACM* 36.1 (Jan. 1993), pp. 78–89. ISSN: 0001-0782. DOI: 10.1145/151233.151240. URL: http://doi.acm.org/10.1145/151233.151240.

[Mil97] Robin Milner. *Speech by Robin Milner on receiving an Honorary Degree from the University of Bologna*. June 9, 1997. URL: http://www.cs.unibo.it/icalp/Lauree_milner.html (visited on 12/29/2014).

[Mil99] Robin Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1999. ISBN: 9780521658690.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. “A Calculus of Mobile Processes, I”. In: *Inf. Comput.* 100.1 (1992), pp. 1–40.

[MS04] Massimo Merro and Davide Sangiorgi. “On asynchrony in name-passing calculi”. In: *Mathematical Structures in Computer Science* 14.5 (2004), pp. 715–767. DOI: 10.1017/S0960129504004323. URL: http://dx.doi.org/10.1017/S0960129504004323.

[Nes00] Uwe Nestmann. “What is a "good" encoding of guarded choice?” In: *Information and Computation* 156.1-2 (2000), pp. 287–319. ISSN: 0890-5401.

[Nes98] Uwe Nestmann. “On the Expressive Power of Joint Input”. In: *Electronic Notes in Theoretical Computer Science* 16.2 (1998). EXPRESS ’98, Fifth International Workshop on Expressiveness in Concurrency (Satellite Workshop of CONCUR ’98), pp. 145–152. ISSN: 1571-0661. DOI: http://dx.doi.org/10.1016/S1571-0661(04)00123-9.

[NP00] Uwe Nestmann and Benjamin C. Pierce. “Decoding Choice Encodings”. In: *Inf. Comput.* 163.1 (2000), pp. 1–59.

[Pad12] Luca Padovani. “On Projecting Processes into Session Types”. In: *Mathematical Structures in Computer Science* 22 (2 2012), pp. 237–289. ISSN: 0960-1295. DOI: 10.1017/S0960129511000405. URL: http://www.di.unito.it/~padovani/Papers/projection.pdf.

[Pal03] Catuscia Palamidessi. “Comparing The Expressive Power Of The Synchronous And Asynchronous Pi-Calculi”. In: *Mathematical Structures in Computer Science*. 2003, pp. 685–719.

[Par08] Joachim Parrow. “Expressiveness of Process Algebras”. In: *Electr. Notes Theor. Comput. Sci.* 209 (2008), pp. 173–186.

[Par14] Joachim Parrow. “General Conditions for Full Abstraction”. In: *Mathematical Structures in Computer Science* (2014). To appear.

[Pér+12] Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. “Linear Logical Relations for Session-Based Concurrency”. In: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012, pp. 539–558. DOI: 10.1007/978-3-642-28869-2_27. URL: http://dx.doi.org/10.1007/978-3-642-28869-2_27.

[Pet12] Kirstin Peters. “Translational expressiveness: comparing process calculi using encodings”. PhD thesis. Technische Universität Berlin, 2012. URL: http://opus4.kobv.de/opus4-tuberlin/frontdoor/index/index/docId/3532.

[Pet62] Carl Adam Petri. “Kommunikation mit Automaten”. PhD thesis. Bonn: Institut für instrumentelle Mathematik, 1962.

[PN12] Kirstin Peters and Uwe Nestmann. “Is It a "Good" Encoding of Mixed Choice?” In: *Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012, pp. 210–224. DOI: 10.1007/978-3-642-28729-9_14. URL: http://dx.doi.org/10.1007/978-3-642-28729-9_14.

[PNG13] Kirstin Peters, Uwe Nestmann, and Ursula Goltz. “On Distributability in Process Calculi”. In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 2013, pp. 310–329. DOI: 10.1007/978-3-642-37036-6_18. URL: http://dx.doi.org/10.1007/978-3-642-37036-6_18.

[PSN11] Kirstin Peters, Jens-Wolfhard Schicke, and Uwe Nestmann. “Synchrony vs Causality in the Asynchronous Pi-Calculus”. In: *arXiv preprint arXiv:1108.4469* (2011).

[PT00] Benjamin C. Pierce and David N. Turner. “Pict: a programming language based on the Pi-Calculus”. In: *Proof, Language, and Interaction, Essays in Honour of Robin Milner*. 2000, pp. 455–494.

[Sew00] Peter Sewell. *Applied π: A Brief Tutorial*. Technical report (University of Cambridge. Computer Laboratory). University of Cambridge, Computer Laboratory, 2000.

[Ste93] William Richard Stevens. *TCP/IP Illustrated (Vol. 1): The Protocols*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1993. ISBN: 0-201-63346-9.

[SW01] Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001. ISBN: 978-0-521-78177-0.

[TCP13] Bernardo Toninho, Luís Caires, and Frank Pfenning. "Higher-Order Processes, Functions, and Sessions: A Monadic Integration." In: *ESOP*. 2013, pp. 350–369.

[Wad12] Philip Wadler. "Propositions as sessions". In: *ACM SIGPLAN Notices*. Vol. 47. 9. ACM. 2012, pp. 273–286.