π with leftovers: a mechanisation in Agda

Uma Zalakain
University of Glasgow, Scotland
u.zalakain.1@research.gla.ac.uk

Ornela Dardha
University of Glasgow, Scotland
ornela.dardha@glasgow.ac.uk

Abstract

The π-calculus is a computational model for communication and concurrency. The *linear* π-calculus is a typed version of the π-calculus where channels must be used exactly once. It is an underlying theoretical and practical framework on top of which more advanced types and theories are built, including *session types* [15, 29, 16]. Linearity is key for type safety in session communication.

We present the *first* full mechanisation in Agda of a π-calculus with *linear*, *graded* and *shared* types, all under the same unified framework. While linearity is key for type safety in communication-centric programming, graded and shared types are needed to model real-world software systems.

We present the syntax, semantics, type system and corresponding type safety properties. For the first time in the π-calculus, we use leftover typing [1] to encode our typing rules in a way that propagates linearity constraints into process continuations. We generalise the algebras on multiplicities, allowing the developer to choose a mix of linear, graded and shared typing. We provide framing, weakening and strengthening proofs that we then use to prove subject congruence. We show that the type system is stable under substitution and prove subject reduction. Our formalisation is fully mechanised in Agda [32].

2012 ACM Subject Classification Theory of computation → Process calculi

Keywords and phrases pi-calculus, linear types, leftover typing, concurrency, mechanisation, Agda

Digital Object Identifier 10.4230/LIPIcs...

Supplement Material `https://github.com/umazalakain/typing-linear-pi`

Funding Supported by the UK EPSRC grant EP/K034413/1, "From Data Types to Session Types: A Basis for Concurrency and Distribution" (ABCD), and by the EU HORIZON 2020 MSCA RISE project 778233 "Behavioural Application Program Interfaces" (BehAPI).

Acknowledgements We want to thank Erika Kreuter, Wen Kokke, James Wood, Guillaume Allais, Bob Atkey, and Conor McBride for their thoughts, patience, work, education and camaraderie.

1 Introduction

We live in a concurrent world where any given state is often interactively computed by a myriad of parties — people, machines, processors, services, networks. As humans, we aim to model, predict, and build such interactive systems; as mathematicians, abstraction is our tool of choice to do so.

The π-calculus [22, 21] is the most successful computational model for communication and concurrency. It abstracts over the details of concurrent processing and boils the interactions down to the sending and receiving of data over communication channels. Notably, it features channel mobility: channels themselves are first order values, and can therefore be transmitted. The π-calculus has been a foundation for the design and implementation of programming languages for concurrency, such as Pict [24] and (more recently) Go [1]. At the state-of-the-art, the π-calculus features a wide plethora of types [17]: basic types, linear types, types for liveness properties (such as deadlock freedom, livelock freedom or termination), and most notably, session types. This makes the π-calculus a fully-fledged tool for modelling and verification of concurrent and distributed systems.

In a parallel track, J.-Y. Girard's development of linear logic [13] in the early '90s opened new avenues in computing science by introducing *linear types* for (functional) programming languages [31, 4]. Linear type systems guarantee that resources are used *exactly once*. Enforcing "no duplicating" and "no discarding" of resources via linearity allows for resource-aware programming and more efficient implementations [31]. Later on, linearity inspired unique types (as in Clean [3]) and ownership types (as in Rust [19]).

The π-calculus also benefited from the linearity wave of the early '90s. Kobayashi et al. [18] defined the linear π-calculus, a typed version of the π-calculus where the linear type system restricts the usage of channels to exactly once. In the meantime, with the rise of *session types* [15, 29, 16] linearity acquired a different flavour in the π-calculus. Session types are a type formalism used in communication-centric programming. In session types, linearity ensures that a channel is owned by exactly one communicating participant, whilst the channel itself is used multiple times as per its session type. Linearity in session types is the key ingredient that guarantees type safety.

Recent work has pushed the linear π-calculus into the spotlight again [18] thanks to an encoding of session types into linear types [7, 6, 8]. This encoding not only has theoretical benefits in terms of the expressivity of session types and the reusability of theoretical results from the linear π-calculus, but is useful at a practical level too. The encoding has been used as a technique to implement session types in mainstream programming languages such as OCaml [23] and Scala [28, 27]. This allows us to use the linear π-calculus as an underlying theoretical and practical framework on top of which session types can be built.

Considering the relevance of the π-calculus and of linearity in modelling concurrent and distributed systems, in this paper we focus our research on both and go beyond.

We present the first full mechanisation in Agda of a π-calculus with linear, graded and shared types, all under the same unified framework.

We do so by defining a specification for an algebra on *multiplicities* (how many times a channel can be used) and use it to apply leftover typing (following Allais [1]) to the π-calculus. The developer is able to choose any mix of algebras for the type system — as long as they comply with the specification. Ultimately, we can exploit this formalisation as a unified framework for type safe distributed modelling and programming with a variety of type systems, and build on top of it more advanced types, theories and languages. In particular, while linearity is needed for type safety in communication-centric programming, graded and shared types are needed to model real-world systems.

1.1 Contributions

1. **Formalisation of the syntax and the semantics of the π-calculus**:
   - **Syntax**: § 2 uses type level de Bruijn indices [9, 10] to introduce a syntax that is *well-scoped by construction*: every free variable is accounted for in the type of the process that uses it.

[1] https://golang.org

- **Semantics**: § 3 provides an operational semantics for the π-calculus, prior to any typing. This operational semantics is given as a reduction relation on processes (§ 3.2). The reduction relation tracks at the type level the channel the communication occurs on, so that this information can then be used to state subject reduction — aka type preservation (Thm. 8). The reduction relation is defined modulo structural congruence (§ 3.1) — a relation that acts as a quotient type that removes unnecessary syntactic minutiae introduced by the syntax of the π-calculus.

2. **Leftover typing for the π-calculus with linear, graded and shared types**: § 4 provides the type system for a resource-aware π-calculus.
   - **Algebras on multiplicities**: The user can provide *resource-aware* algebras, which are then applied to the type system (§ 4.1). Linear types, graded types and shared types are all instances of the same resource-aware algebra, giving rise to three different type systems under the same framework. Any *partial commutative monoid* that is *functional*, *cancellative* and has a *minimal element* is a valid such algebra.
     Multiple algebras can be simultaneously used in a single type system — usage contexts keep information about what algebra to use on which element (§ 4.2). This allows for type systems combining linear, graded and shared types.
   - **π-calculus with leftovers**: Our type system uses *leftover typing* to model the resource-aware π-calculus (§ 4.3). This approach adds a leftover usage context to the typing judgements. Typing derivations take the resources of their input usage context, consume some of them, and leave the rest as leftovers in the output usage context.
     Benefits of typing with leftovers include: removing the need for extrinsic context splits, which are rendered unnecessary; and for the first time we can state theorems like **weakening** (Thm. 2) and **strengthening** (Thm. 3) in the linear π-calculus.
3. **Machine verified proofs of subject reduction and auxiliary theorems**: § 5 details the type safety properties of our π-calculus with leftovers. Leftover typing allows *framing* (Thm. 1), *weakening* (Thm. 2) and *strengthening* (Thm. 3) theorems to be stated. We use these to prove *subject congruence* (Thm. 5), and together with *substitution* (Thm. 7), to prove *subject reduction* (Thm. 8).
4. **Fully mechanised formalisation available in Agda**: This work has been fully mechanised in Agda and is publicly available in our GitHub repository [32].

1.2 Notation

$$\genfrac{}{}{}{0}{}{\mathbb{N} : \mathrm{SET}}\ \textsc{Nat} \qquad\qquad \frac{}{0 : \mathbb{N}} \qquad\qquad \frac{n : \mathbb{N}}{1{+}n : \mathbb{N}}$$

Figure 1 Notation used in this paper.

Fig. 1 illustrates the notation used in this paper. Data type definitions ($\mathbb{N}$) use double lines and index-free synonyms (Nat) as rule names (for ease of reference). We otherwise use constructor names (0 and $1{+}$) to name our typing rules. Universe levels and universe polymorphism are omitted for brevity — all our types are of type SET. Implicit arguments are mentioned by type definitions but omitted by constructors.

We use colours to further distinguish the different entities in this paper. TYPES are blue violet (uppercased, with indices as subscripts), constructors are burnt orange, functions are olive green, variables are black, and some constructor names are overloaded — and disambiguated by context.

2 Syntax

$$\frac{n : \mathbb{N}}{\mathrm{VAR}_n : \mathrm{SET}}\ \textsc{Var} \qquad \frac{n : \mathbb{N}}{0 : \mathrm{VAR}_{1+n}} \qquad \frac{x : \mathrm{VAR}_n}{1{+}x : \mathrm{VAR}_{1+n}} \qquad \frac{n : \mathbb{N}}{\mathrm{PROCESS}_n : \mathrm{SET}}\ \textsc{Process}$$

$$\begin{array}{rl}
P, Q ::= & \mathbb{0} \\
\mid & (\nu\ x)\ P \\
\mid & P \parallel Q \\
\mid & x\ (\ y\ )\ .\ P \\
\mid & x\ \langle\ y\ \rangle\ .\ P
\end{array}
\qquad
\begin{array}{rll}
\mathrm{PROCESS}_n ::= & \mathbb{0}_n & \text{(inaction)} \\
\mid & \nu\ \mathrm{PROCESS}_{1+n} & \text{(restriction)} \\
\mid & \mathrm{PROCESS}_n \parallel \mathrm{PROCESS}_n & \text{(parallel)} \\
\mid & \mathrm{VAR}_n\ ()\ \mathrm{PROCESS}_{1+n} & \text{(input)} \\
\mid & \mathrm{VAR}_n\ \langle\ \mathrm{VAR}_n\ \rangle\ \mathrm{PROCESS}_n & \text{(output)}
\end{array}$$

(a) Ill-scoped grammar with names. **(b)** Well-scoped grammar with type-level de Bruijn indices.

Figure 2

Let $x, y, \ldots$ range over *variables* (including channel names) and $P, Q, \ldots$ over processes. The syntax of the π-calculus [26] is given by the grammar in Fig. 2a. Process $\mathbb{0}$ denotes the terminated process, where no operations can further occur. Process $(\nu\ x)\ P$ creates a new channel x bound with scope P. Process $P \parallel Q$ is the parallel composition of processes P and Q. Processes $x\ (\ y\ )\ .\ P$ and $x\ \langle\ y\ \rangle\ .\ P$ denote respectively the input and output processes of a variable y over a channel x, with continuation P.

Scope restriction $(\nu\ x)\ P$ and input $x\ (\ y\ )\ .\ P$ are *binders*, they are the only constructs which introduce bound names — x and y in P, respectively. In order to mechanise the π-calculus syntax in Agda, we need to deal with bound names in continuation processes. Names are cumbersome to mechanise: inserting a new variable into a context means proving that its name differs from all other variable names in context. Instead, we use de Bruijn indices [9], where a natural number n (aka *index*) is used to refer to the variable introduced n binders ago — hence binders no longer introduce names.

▶ **Example 1.**

$$\begin{array}{llllllll}
(\nu\ x) & (x\ (\ y\ )\ . & y\ \langle\ a\ \rangle\ . & \mathbb{0} \parallel (\nu\ y) & (x\ \langle\ y\ \rangle\ . & y\ (\ z\ )\ . & \mathbb{0})) & \text{names} \\
\nu & (0\ () & 0\ \langle\ a\ \rangle & \mathbb{0} \parallel \nu & (1\ \langle\ 0\ \rangle & 0\ () & \mathbb{0})) & \text{de Bruijn indices}
\end{array}$$

That is, terms at different *depths* use different indices to refer to the same binding.

A variable reference occurring under n binders can refer to n distinct variables. References outside of that range are meaningless. It is useful to rule out these ill-scoped nonsensical terms syntactically. In Fig. 2b, we do so by introducing the indexed family of types [10] VAR_n: for all naturals n, the type VAR_n has n distinct elements. We index processes according to their *depth*: for all naturals n, a process of type $\mathrm{PROCESS}_n$ contains variables that can refer to n distinct elements. Every time we go under a binder, we increase the index of the continuation process, allowing the variable references within to refer to one more thing.

3 Semantics

Thanks to our well-scoped grammar in Fig. 2b, the semantics of our language can now be defined on the totality of the syntax. We start by defining structural congruence in § 3.1 and then the reduction relation in § 3.2.

3.1 Structural Congruence

We define the base cases of a structural congruence relation $\textsc{StructCong}\ \equiv$ in Fig. 3. Its congruent equivalence closure $\textsc{Equals}\ =$ is defined in App. B.

$$\frac{}{P \equiv Q : \mathrm{SET}}\ \textsc{StructCong} \qquad \frac{}{\text{comp-assoc} : P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R}$$

$$\frac{}{\text{comp-sym} : P \parallel Q \equiv Q \parallel P} \qquad \frac{}{\text{comp-end} : P \parallel \mathbb{0}_n \equiv P} \qquad \frac{}{\text{scope-end} : \boldsymbol{\nu}\ \mathbb{0}_{1+n} \equiv \mathbb{0}_n}$$

$$\frac{uQ : \mathrm{UNUSED}_0\ Q}{\text{scope-ext} : \boldsymbol{\nu}\ (P \parallel Q) \equiv (\boldsymbol{\nu}\ P) \parallel \mathrm{lower}_0\ \ Q\ uQ} \qquad \frac{}{\text{scope-comm} : \boldsymbol{\nu}\ \boldsymbol{\nu}\ P \equiv \boldsymbol{\nu}\ \boldsymbol{\nu}\ \mathrm{swap}_0\ P}$$

Figure 3 Base cases of structural congruence.

The first three rules (comp−*) state associativity, symmetry, and $\mathbb{0}$ as being the neutral element of parallel composition, respectively. The last three (scope−*) state garbage collection, scope extrusion and commutativity of restrictions, respectively.

In scope-ext the type $\mathrm{UNUSED}_i\ Q$ is an inductive proof asserting that the variable index i does not appear neither in the inputs nor in the outputs of Q. The function $\mathrm{lower}_i\ Q\ uQ$ traverses Q decrementing every index greater than i. In scope-comm the function $\mathrm{swap}_i\ P$ traverses P (of type $\mathrm{PROCESS}_{1+1+n}$) and swaps variable references i and $1{+}i$. In all the above, i is incremented every time we go under a binder.

3.2 Reduction Relation

The operational semantics of the π-calculus is defined as a reduction relation $\textsc{Reduces}\ \longrightarrow_c$ indexed by the channel c on which communication occurs (Fig. 4). We keep track of channel c so we can state subject reduction (Thm. 8). We distinguish between channels that are created inside of the process (internal) and channels that are created outside (external i, where i is the index of the channel variable).

In rule comm, parallel processes reduce when they communicate over a common channel with index i. As a result of that communication, the continuation of the input process P has all the references to its most immediate variable substituted with references to $1{+}j$, the variable sent by the output process $i\ \langle\ j\ \rangle\ Q$. After substitution, $P\ [\ 0 \mapsto 1{+}j\ ]$ is *lowered* — all variable references are decreased by one (we apply Lem. 1 in App. A to obtain a proof $\mathrm{UNUSED}_0\ \ (P\ [\ 0 \mapsto 1{+}j\ ])$). Reduction is closed under parallel composition (rule par), restriction (rule res) and structural congruence (rule struct) — notably, not under input nor output, as doing so would not preserve the sequencing of actions [26].

In res, we use dec to decrement the index of channel c as we wrap processes P and Q inside a binder. This dec function saturates at internal: wrapping an internally reducing process with a binder will result in an internally reducing process.

$$\frac{n : \mathbb{N}}{\mathrm{CHANNEL}_n : \mathrm{SET}}\ \textsc{Channel} \qquad \frac{}{\mathrm{internal} : \mathrm{CHANNEL}_n} \qquad \frac{i : \mathrm{VAR}_n}{\mathrm{external}\ i : \mathrm{CHANNEL}_n}$$

$$\frac{c : \mathrm{CHANNEL}_n \qquad P\ Q : \mathrm{PROCESS}_n}{P \longrightarrow_c Q : \mathrm{SET}}\ \textsc{Reduces}$$

$$\frac{i\ j : \mathrm{VAR}_n \qquad P : \mathrm{PROCESS}_{1+n} \qquad Q : \mathrm{PROCESS}_n}{\mathrm{comm} : i\ ()\ P \parallel i\ \langle\ j\ \rangle\ Q \longrightarrow_{\mathrm{external}\ i} \mathrm{lower}_0\ (P\ [\ 0 \mapsto 1{+}j\ ])\ uP' \parallel Q}$$

$$\frac{red : P \longrightarrow_c P'}{\mathrm{par}\ red : P \parallel Q \longrightarrow_c P' \parallel Q} \qquad \frac{red : P \longrightarrow_c Q}{\mathrm{res}\ red : \boldsymbol{\nu}\ P \longrightarrow_{\mathrm{dec}\ c} \boldsymbol{\nu}\ Q}$$

$$\frac{eq : P =_r P' \qquad red : P' \longrightarrow_c Q}{\mathrm{struct}\ eq\ red : P \longrightarrow_c Q}$$

Figure 4 Operational semantics indexed by the channel over which reduction occurs.

4 Resource-aware Type System

In § 4.1 we characterise a usage algebra for our type system. It defines how resources are *split* in parallel composition and *consumed* in input and output. We define typing and usage contexts in § 4.2. We provide a type system for a resource-aware π-calculus in § 4.3.

4.1 Multiplicities and Capabilities

In the linear π-calculus each channel has an input and an output *capability*, and each capability has a given *multiplicity* of 0 (exhausted) or 1 (available). We generalise over this notion by defining an algebra that is satisfied by linear, graded and shared types alike.

▶ **Definition 1** (Usage algebra). A *usage algebra* is a ternary relation $x \coloneqq y \cdot z$, that is *partial*, *functional*, *cancellative*, *associative*, and *commutative*. It has a *neutral element* $0\cdot$ that is absorbed on either side, and that is also *minimal*. It has an element $1\cdot$ that is used to count inputs and outputs. Fig. 5 uses a terniary relation to model such an algebra as a record $\mathrm{ALGEBRA}_C$ on a carrier C.

▶ Note 2. We will often work with pairs of input and output multiplicities. We use the notation C^2 to stand for a $\mathrm{C}\times\mathrm{C}$ tuple of input and output multiplicities, respectively. We use $x \coloneqq y \cdot^2 z$ to stand for a monoidal operation on pairs of multiplicities – where the algebraic laws are lifted element-wise.

Henceforth, we use $\ell_\varnothing$ to denote the pair $0\ ,\ 0$ for a channel which cannot be used any further, ℓ_i for the pair $1\ ,\ 0$ for a channel to be used in input exactly once, ℓ_o for $0\ ,\ 1$ for a channel to be used in output exactly once, and $\ell_\#$ for $1\ ,\ 1$, for a channel to be used exactly once in input and exactly once in output. This notation was originally used in the linear π-calculus [18, 26].

Linear, graded and shared types are all defined as an instance of our usage algebra. Their use in typing derivations is illustrated in Ex. 5.

$$
\begin{array}{lll}
0\cdot & : & C \\
1\cdot & : & C \\
\_ \coloneqq \_ \cdot \_ & : & C \to C \to C \to \mathrm{SET} \\
\cdot\text{-compute} & : \forall yz & \to \mathrm{DEC}\ (\exists x\ (x \coloneqq y \cdot z)) \\
\cdot\text{-unique} & : \forall x x' y z & \to x' \coloneqq y \cdot z \to x \coloneqq y \cdot z \to x' \equiv x \\
\cdot\text{-unique}^{l} & : \forall x y y' z & \to x \coloneqq y' \cdot z \to x \coloneqq y \cdot z \to y' \equiv y \\
\cdot\text{-min}^{l} & : \forall yz & \to 0\cdot \coloneqq y \cdot z \to y \equiv 0\cdot \\
\cdot\text{-id}^{l} & : \forall x & \to x \coloneqq 0\cdot \cdot x \\
\cdot\text{-comm} & : \forall xyz & \to x \coloneqq y \cdot z \to x \coloneqq z \cdot y \\
\cdot\text{-assoc} & : \forall xyzuv & \to x \coloneqq y \cdot z \to y \coloneqq u \cdot v \to \exists w\ (x \coloneqq u \cdot w \times w \coloneqq v \cdot z)
\end{array}
$$

Figure 5 Usage algebra $\mathrm{ALGEBRA}_C$ on a carrier C. We use $\forall$ for universal quantification. The dependent product $\exists$ uses the value of its first argument in the type of its second. The type $\mathrm{DEC}\ P$ is a witness of either P or $P \to \bot$, where $\bot$ is the empty type with no constructors.

Linear The carrier is a type with two trivial constructors zero and one. The monoidal operation has the element zero as neutral on both sides, and the element one splitting into one and zero (or zero and one), and is uninhabited otherwise. All the properties of the algebra follow trivially. As an example, the multiplicity pair ℓ_i is represented as one , zero.

Graded The carrier is the type of natural numbers. The element $0\cdot$ corresponds to 0, and $1\cdot$ to 1. The partial monoid $x \coloneqq y \cdot z$ is defined exactly when $x = y + z$. All other properties of the algebra follow from the algebraic rules for the addition of natural numbers. As an example, we use 5 , 3 to represent the multiplicities of a channel that is able to input 5 times and output 3 times.

Shared The carrier is a type with a single trivial constructor ω. Both $0\cdot$ and $1\cdot$ are instantiated to ω. The relation $\omega \coloneqq \omega \cdot \omega$ is inhabited. All the properties of the algebra follow trivially. Shared channels have multiplicities ω , ω, which they will forever preserve.

4.2 Typing Contexts

We use indexed sets of usage algebras to allow several usage algebras to coexist in our type system with leftovers (§ 4.3).

▶ **Definition 2** (Indexed set of usage algebras)**.** An *indexed set of usage algebras* is a nonempty type IDX of indices together with an interpretation USAGE of indices into types, and an interpretation ALGEBRAS of indices into usage algebras (Fig. 6).

We keep typing contexts (PRECTX, in Fig. 7) and usage contexts (CTX in Fig. 8) separate. The former are preserved throughout typing derivations; the latter are transformed as a result of input, output, and context splits.

▶ **Definition 3** (Type and PreCtx: types and typing contexts)**.** A *type* is either a unit type ($\mathbb{1}$), a base type (B[n]) or a channel type (C[t ; x]) (Fig. 7).

The unit type $\mathbb{1}$ serves as a proof of inhabitance for types. The base type B[n] uses natural numbers as placeholders for types (the host language can then interpret the former

$$\begin{array}{ll}
\mathrm{IDX} & : \mathrm{SET} \\
\exists\mathrm{IDX} & : \mathrm{IDX} \\
\mathrm{USAGE} & : \mathrm{IDX} \rightarrow \mathrm{SET} \\
\mathrm{ALGEBRAS} & : (idx : \mathrm{IDX}) \rightarrow \mathrm{ALGEBRA}_{\mathrm{USAGE}_{idx}}
\end{array}$$

Figure 6 Indexed set of usage algebras.

$$\frac{}{\mathrm{TYPE} : \mathrm{SET}}\textsc{Type} \qquad \frac{}{\mathbb{1} : \mathrm{TYPE}} \qquad \frac{n : \mathbb{N}}{\mathrm{B}[\,n\,] : \mathrm{TYPE}} \qquad \frac{t : \mathrm{TYPE} \qquad \begin{array}{c} idx : \mathrm{IDX} \\ x : \mathrm{USAGE}^2_{idx} \end{array}}{\mathrm{C}[\,t\,;\,x\,] : \mathrm{TYPE}}$$

$$\frac{n : \mathbb{N}}{\mathrm{PRECTX}_n : \mathrm{SET}}\textsc{PreCtx} \qquad \frac{}{[] : \mathrm{PRECTX}_0} \qquad \frac{\gamma : \mathrm{PRECTX}_n \qquad t : \mathrm{TYPE}}{\gamma\,,\,t : \mathrm{PRECTX}_{1+n}}$$

Figure 7 Types and length-indexed typing contexts.

into the latter — e.g. 0 as booleans, 1 as natural numbers) so that we avoid having to deal with universe polymorphism. The type $\mathrm{C}[\,t\,;\,x\,]$ of a channel determines what type of data t and what usage annotations x are sent over that channel. Notice that this notation is similar to Kobayashi's $[t]\mathbf{chan}_{(i^y,o^z)}$, where y, z are the multiplicities of input and output, respectively [17].

A *typing context* (last row in Fig. 7) is a length-indexed list of types, and is either empty $[]$ or the result of appending a type t to an existing context γ.

▶ **Definition 4** (Ctx: usage contexts). A *usage context* is a context CTX_{idxs} of usage annotations that is indexed by a length-indexed context of indices IDXS_n (Fig. 8).

A usage context is either empty $[]$ or the result or appending a usage annotation x to the existing context Γ.

$$\frac{n : \mathbb{N}}{\mathrm{IDXS}_n : \mathrm{SET}}\textsc{Idxs} \qquad \frac{}{[] : \mathrm{IDXS}_0} \qquad \frac{idxs : \mathrm{IDXS}_n \qquad idx : \mathrm{IDX}}{idxs\,,\,idx : \mathrm{IDXS}_{1+n}}$$

$$\frac{idxs : \mathrm{IDXS}_n}{\mathrm{CTX}_{idxs} : \mathrm{SET}}\textsc{Ctx} \qquad \frac{}{[] : \mathrm{CTX}_{[]}} \qquad \frac{\Gamma : \mathrm{CTX}_{idxs} \qquad x : \mathrm{USAGE}^2_{idx}}{\Gamma\,,\,x : \mathrm{CTX}_{idxs\,,\,idx}}$$

Figure 8 Length-indexed context of carrier indices with a context of usage annotations on top.

▶ Note 3. We lift the monoidal operation on usage annotations $x \coloneq y \cdot^2 z$ to a monoidal operation $\Gamma \coloneq \Delta \otimes \Xi$ on usage contexts that have a common underlying context of indices. All the properties of the partial monoid are lifted element-wise.

4.3 Typing with Leftovers

We use *leftover typing* [1] for our type system, an approach that, in addition to the usual typing context PRECTX_n and (input) usage context CTX_{idxs}, adds an extra *output usage context* CTX_{idxs} to the typing rules.

This output context contains the *leftovers* (the unused multiplicities) of the process being typed. These leftovers can then be used as an input to another typing derivation. Leftover typing presents three main advantages:

- **Encapsulation**: The transformations to the usage context (consequence of receiving and sending over a channel) are encapsulated into a variable reference judgement with both an input and an output usage context.
- **No extensional context split**: No need for an extensional context splitting proof for parallel composition. Such a proof would need to construct a witness $\Gamma_m \coloneqq \Gamma_l \otimes \Gamma_r$, where $\Gamma_l \vdash P$, $\Gamma_r \vdash Q$ and $\Gamma_m \vdash P \parallel Q$. This could be obtained through automated search — but it is better yet not to need it. Leftover typing uses the leftovers of P to type Q.
- **New theorems**: For the first time, theorems like **framing** (Thm. 1), **weakening** (Thm. 2) and **strengthening** (Thm. 3) can be stated within the linear π-calculus thanks to the presence of a leftover context.

Our type system with leftovers is composed of two typing relations: one for variable references (Def. 5) and one for processes (Def. 6). Both relations are indexed by a typing context γ, an input usage context Γ, and an output usage context Δ (the leftovers).

The typing judgement for variables $\gamma \,;\, \Gamma \ni_i t \,;\, y \triangleright \Delta$ asserts that "index i in typing context γ is of type t, and subtracting y at position i from input usage context Γ results in leftovers Δ". The typing judgement for processes $\gamma \,;\, \Gamma \vdash P \triangleright \Delta$ asserts that "process P is well typed under typing context γ, usage input context Γ and leftovers Δ".

▶ **Definition 5** (VarRef: typing variable references)**.** The VarRef typing relation for variable references is presented in Fig. 9.

$$\frac{\gamma : \mathrm{PRECTX}_n \quad \begin{matrix} idxs : \mathrm{IDXS}_n \\ \Gamma : \mathrm{CTX}_{idxs} \end{matrix} \quad i : \mathrm{VAR}_n \quad t : \mathrm{TYPE} \quad \begin{matrix} idx : \mathrm{IDX} \\ y : \mathrm{USAGE}^2_{idx} \end{matrix} \quad \Delta : \mathrm{CTX}_{idxs}}{\gamma \,;\, \Gamma \ni_i t \,;\, y \triangleright \Delta : \mathrm{SET}} \; \textsc{VarRef}$$

$$\frac{x \coloneqq y \cdot^2 z}{0 : \gamma \,,\, t \,;\, \Gamma \,,\, x \ni_0 t \,;\, y \triangleright \Gamma \,,\, z} \qquad \frac{loc_i : \gamma \,;\, \Gamma \ni_i t \,;\, x \triangleright \Delta}{1\!+\, loc_i : \gamma \,,\, t' \,;\, \Gamma \,,\, x' \ni_{1+i} t \,;\, x \triangleright \Delta \,,\, x'}$$

Figure 9 Typing rules for variable references.

The base case 0 splits the usage annotation x of type USAGE_{idx} into y and z (the leftovers). This splitting $x \coloneqq y \cdot^2 z$ is as per the usage algebra provided by the developer for the index idx. We use the *functionality* of the monoidal relation to alleviate the user from the proof burden $x \coloneqq y \cdot^2 z$. The inductive case $1+$ appends the type t' to the typing context, and the usage annotation x' to both the input and output usage contexts.

▶ **Example 4** (Variable reference)**.** Let us illustrate the use of variable reference relations with an example in Listing 1.

Listing 1 Typing variable reference $1{+}0$ with type $\mathrm{C}[\ \mathbb{1}\ ;\ \ell_{\mathrm{i}}\ ]$ and usage ℓ_{i}.

```
_ : [] , C[ 𝟙 ; ℓi ] , 𝟙 ; [] , ℓ# , ℓ# ∋1+ 0 C[ 𝟙 ; ℓi ] ; ℓi ▹ [] , ℓo , ℓ#
_ = 1+ 0
```

The underscore _ introduces an anonymous declaration immediately followed by its definition. The constructor $1{+}$ steps under the outermost variable in the context, preserving its usage annotation $\ell_{\#}$ from input to output. The constructor 0 asserts that the next variable is of a channel type $\mathrm{C}[\ \mathbb{1}\ ;\ \ell_{\mathrm{i}}\ ]$, and that we can split its input annotation $\ell_{\#}$, taking away ℓ_{i} and leaving ℓ_{o} as the leftover, to be used later on.

▶ **Definition 6** (Types: typing processes)**.** The Types typing relation for π-calculus processes is presented in Fig. 10.

$$
\begin{array}{c}
\gamma : \mathrm{PRECTX}_n \qquad \begin{array}{l} idxs : \mathrm{IDXS}_n \\ \Gamma : \mathrm{CTX}_{idxs} \end{array} \qquad P : \mathrm{PROCESS}_n \qquad \Delta : \mathrm{CTX}_{idxs} \\
\hline\hline
\gamma\ ;\ \Gamma \vdash P \triangleright \Delta : \mathrm{SET}
\end{array}\ \textsc{Types}
$$

$$
\frac{}{\mathrm{end} : \gamma\ ;\ \Gamma \vdash \mathbb{0} \triangleright \Gamma}
\qquad
\frac{\begin{array}{c} t : \mathrm{TYPE} \qquad x : \mathrm{USAGE}^2_{idx} \qquad y : \mathrm{USAGE}_{idx'} \\ cont : \gamma\ ,\ \mathrm{C}[\ t\ ;\ x\ ]\ ;\ \Gamma\ ,\ (y\ ,\ y) \vdash P \triangleright \Delta\ ,\ \ell_{\varnothing} \end{array}}{\mathrm{chan}\ t\ x\ y\ cont : \gamma\ ;\ \Gamma \vdash \boldsymbol{\nu}\ P \triangleright \Delta}
$$

$$
\frac{\begin{array}{l} chan_i : \gamma\ ;\ \Gamma \ni_i \mathrm{C}[\ t\ ;\ x\ ]\ ;\ \ell_{\mathrm{i}} \triangleright \Xi \\ cont\ : \gamma\ ,\ t\ ;\ \Xi\ ,\ x \vdash P \triangleright \Theta\ ,\ \ell_{\varnothing} \end{array}}{\mathrm{recv}\ chan_i\ cont : \gamma\ ;\ \Gamma \vdash i\ (\,)\ P \triangleright \Theta}
\qquad
\frac{\begin{array}{l} chan_i : \gamma\ ;\ \Gamma \ni_i \mathrm{C}[\ t\ ;\ x\ ]\ ;\ \ell_{\mathrm{o}} \triangleright \Delta \\ loc_j\ : \gamma\ ;\ \Delta \ni_j t\ ;\ x \triangleright \Xi \\ cont\ : \gamma\ ;\ \Xi \vdash P \triangleright \Theta \end{array}}{\mathrm{send}\ chan_i\ loc_j\ cont : \gamma\ ;\ \Gamma \vdash i\ \langle\ j\ \rangle\ P \triangleright \Theta}
$$

$$
\frac{\begin{array}{l} l\ : \gamma\ ;\ \Gamma \vdash P \triangleright \Delta \\ r : \gamma\ ;\ \Delta \vdash Q \triangleright \Xi \end{array}}{\mathrm{comp}\ l\ r : \gamma\ ;\ \Gamma \vdash P \parallel Q \triangleright \Xi}
$$

Figure 10 Leftover typing for a resource-aware type system.

The inaction process in rule end does not change usage annotations. The scope restriction in rule chan expects three arguments: the type t of data being transmitted; the usage annotation x of what is being transmitted; and the multiplicity y given to the channel itself. This multiplicity y is used for both input and output, so that they are balanced. The continuation process P is provided with the new channel with usage annotation $y\ ,\ y$, which it must completely exhaust.

The input process in rule recv requires a channel $chan_i$ at index i with usage ℓ_{i} available, such that data with type t and usage x can be sent over it. Note that the index i is used in the syntax of the typed process. We use the leftovers Ξ to type the continuation process, which is also provided with the received element — of type t and multiplicity x at index 0. The received element x must be completely exhausted by the continuation process.

Similarly to input, the output process in rule send requires a channel $chan_i$ at index i with usage ℓ_{o} available, such that data with type t and usage x can be sent over it. We use the leftover context Δ to type the continuation process, which needs an element loc_j at index j with type t and usage x, as per the type of the channel $chan_i$. The leftovers Ξ are used to type the continuation process. Note that both indices i and j are used in the syntax of the typed process.

Parallel composition in rule comp uses the leftovers of the left-hand process to type the right-hand process. By keeping track in the typing derivation of P of the resources P uses, we can use them to type Q and save the user from having to provide an extrinsic proof of context split.

▶ **Example 5** (Typing derivation). We will now illustrate a typing derivation of the process p using linear and shared types (Listing 2).

Let ω be a shortcut for $1\cdot$, $1\cdot$ in an algebra of shared types. Let ϵ be the empty usage context — where all usage annotations are $0\cdot$, $0\cdot$. To type p, we need a channel over which we can transmit another channel. We therefore create a new channel with usage annotations $1\cdot$, $1\cdot$ capable of transmitting data of type $C[\ \mathbb{1}\ ;\ \omega\ ]$, and declare (with ℓ_i) that we only transmit input capabilities over it (line 7). The left-hand process is typed following its syntax (line 8). On the right-hand we need the channel that is going to be sent (line 9). The type $\mathbb{1}$ of this channel is entirely determined by the type $C[\ \mathbb{1}\ ;\ \omega\ ]$ of the channel over which it is going to be transmitted. Its usage annotation is $1\cdot$, $1\cdot$: we will transmit $1\cdot$, $0\cdot$ and save $0\cdot$, $1\cdot$ for the continuation process.

Listing 2 Typing a process that sends part of one channel over another.

```
1   p : PROCESS₁₊zero
2   p = ν
3         (0 () (0 () 𝟘) ‖
4         ν (1+ 0 ⟨ 0 ⟩ (0 ⟨ 1+ 1+ 0 ⟩ ) 𝟘))
5
6   _ : [] , 𝟙 ; [] , ω ⊢ p ▹ epsilon
7   _ = chan C[ 𝟙 ; ω ] ℓᵢ 1· (comp
8         (recv 0 (recv 0 end))
9         (chan 𝟙 ω 1· (send (1+ 0) 0 (send 0 (1+ 1+ 0) end))))
```

5 Type Safety

Framing asserts that the well-typedness of a process is independent of its leftover resources.

▶ **Theorem 1** (Framing). Let P be well typed in $\gamma\ ;\ \Gamma_l \vdash P \triangleright \Xi_l$. Let Δ be such that $\Gamma_l \coloneqq \Delta \otimes \Xi_l$. Let Γ_r and Ξ_r be arbitrary contexts such that $\Gamma_r \coloneqq \Delta \otimes \Xi_r$.

Then $\gamma\ ;\ \Gamma_r \vdash P \triangleright \Xi_r$.

Weakening states that inserting a new variable into the context preserves the well-typedness of a process as long as the usage annotation of the inserted variable is preserved as a leftover. Let ins_i insert an element into a context at position i — for simplicity, we use it both to insert types into typing contexts and usage annotations into usage contexts.

▶ **Theorem 2** (Weakening). Let P be well typed in $\gamma\ ;\ \Gamma \vdash P \triangleright \Xi$. Then, lifting every variable greater than or equal to i in P is well typed in $\mathrm{ins}_i\ t\ \gamma\ ;\ \mathrm{ins}_i\ x\ \Gamma \vdash \mathrm{lift}_i\ P \triangleright \mathrm{ins}_i\ x\ \Xi$.

Strengthening states that removing an unused variable preserves the well-typedness of a process. Let del_i delete the element at position i from a context — for simplicity, we use it both to delete types from typing contexts and usage annotations from usage contexts.

▶ **Theorem 3** (Strengthening). Let P be well typed in $\gamma\,;\,\Gamma \vdash P \triangleright \Xi$. Let i be a variable not in P, such that $\mathrm{UNUSED}_i\ P$. Then lowering every variable greater than i in P is well typed in $\mathrm{del}_i\ \gamma\,;\,\mathrm{del}_i\ \Gamma \vdash \mathrm{lower}_i\ P \triangleright \mathrm{del}_i\ \Xi$.

▶ Remark 6. The *unusedness* predicate cannot depend on the preservation of usage annotations by a process: the process could preserve such annotations, yet sent a $\ell_\varnothing$ usage of those variables over a channel.

Swapping two variables preserves the well-typedness of a process. For simplicity, we extend swap_i introduced in § 3.1 to swap types in typing contexts and usage annotations in usage contexts.

▶ **Theorem 4** (Swapping). Let P be well typed in $\gamma\,;\,\Gamma \vdash P \triangleright \Xi$. Then, $\mathrm{swap}_i\ \gamma\,;\,\mathrm{swap}_i\ \Gamma \vdash \mathrm{swap}_i\ P \triangleright \mathrm{swap}_i\ \Xi$.

Proof (Sketch). All the above theorems are proved by induction on Process and Var. For details, refer to our mechanisation in Agda [32]. ◀

Subject Congruence states that applying structural congruence rules to a well typed process preserves its well-typedness.

▶ **Theorem 5** (Subject congruence). If $P = Q$ and $\gamma\,;\,\Gamma \vdash P \triangleright \Xi$, then $\gamma\,;\,\Gamma \vdash Q \triangleright \Xi$.

Proof (Sketch). The proof is by induction on Equals $=$. For the nontrivial base cases and their symmetric variants, we apply a combination of Thm. 1, Thm. 3, Thm. 2 and Thm. 4. Further details can be found in App. B and the full proof in [32]. ◀

Substitution Thm. 6 proves a generalised version of substitution, where the substitution $P\,[\,i \mapsto j\,]$ is on any variable i. Thm. 7 instantiates the generalised version to the concrete case where i is the most recently introduced variable 0, as required by subject reduction.

▶ **Theorem 6** (Generalised substitution). Let process P be well typed in $\gamma\,;\,\Gamma_i \vdash P \triangleright \Psi_i$. Then, there must exist some Γ, Ψ, Γ_j and Ψ_j such that:

- $\gamma\,;\,\Gamma_i \ni_i t\,;\,m \triangleright \Gamma$
- $\gamma\,;\,\Gamma_j \ni_j t\,;\,m \triangleright \Gamma$
- $\gamma\,;\,\Psi_i \ni_i t\,;\,n \triangleright \Psi$
- $\gamma\,;\,\Psi_j \ni_j t\,;\,n \triangleright \Psi$

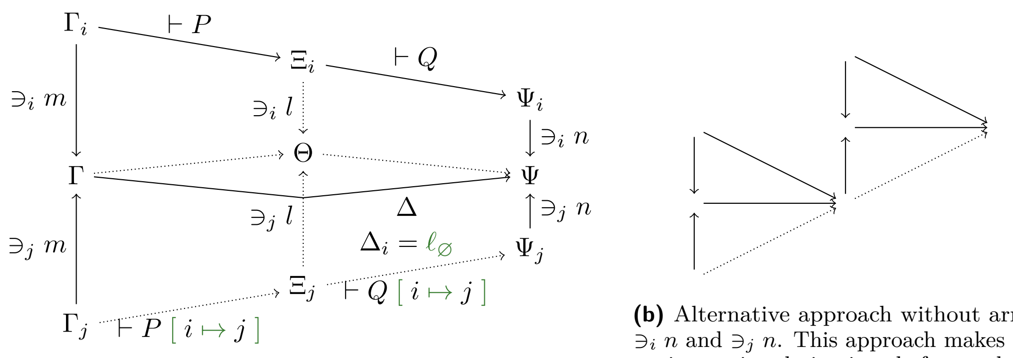
Let Γ and Ψ be such that $\Gamma \coloneqq \Delta \otimes \Psi$ for some Δ. Let Δ at position i have usage $\ell_\varnothing$, meaning all consumption from m to n must happen in P. Then substituting i with j in P will be well typed in $\gamma\,;\,\Gamma_j \vdash P\,[\,i \mapsto j\,] \triangleright \Psi_j$.

Proof (Sketch). By induction on $\gamma\,;\,\Gamma_i \vdash P \triangleright \Psi_i$. Full details are found in App. C.
- Constructor end: observe that $\Gamma_i \equiv \Psi_i$ and thus $\Gamma_j \equiv \Psi_j$.
- Constructor chan: we proceed inductively.
- Constructors recv, send and comp: we must find Θ in Fig. 11a and split the diagram along its vertical axis to proceed by induction. ◀

▶ **Theorem 7** (Substitution). Let P be well typed in $\gamma\ ,\ t\,;\,\Gamma\ ,\ m \vdash P \triangleright \Psi\ ,\ \ell_\varnothing$. Let $\gamma\,;\,\Psi \ni_j t\,;\,m \triangleright \Xi$. Then, we can substitute the variable references to 0 in P with $1{+}j$ so that the result is well typed in $\gamma\ ,\ t\,;\,\Gamma\ ,\ m \vdash P\,[\,0 \mapsto 1{+}j\,] \triangleright \Xi\ ,\ m$.

Proof (Sketch). We use framing to derive $\gamma\,;\,\Gamma \ni_j t\,;\,m \triangleright \Theta$ and $\gamma\ ,\ t\,;\,\Theta\ ,\ m \vdash P \triangleright \Xi\ ,\ \ell_\varnothing$ for some Θ. Then, we use these to apply Thm. 6. ◀

▶ Note 7. We use $\Gamma \ni_i x \triangleright \Delta$ to stand for $\gamma\,;\,\Gamma \ni_i t\,;\,x \triangleright \Delta$ for some γ and t.



(a) Diagrammatic representation of the comp case for substitution. Continuous lines represent known facts, dotted lines proof obligations.

(b) Alternative approach without arrows $\ni_i n$ and $\ni_j n$. This approach makes composing typing derivations before and after substitution more difficult.

Figure 11 Substitution with and without the *adjustment* arrows $\ni_i n$ and $\ni_j n$

Subject Reduction states that if P is well typed and it reduces to Q, then Q is well typed. In the π-calculus we distinguish between a reduction $P \longrightarrow_{\text{internal}} Q$ on a channel internal to P, and a reduction $P \longrightarrow_{\text{external } i} Q$ on a channel i external to P (refer to § 3.2).

▶ **Theorem 8** (Subject reduction)**.** Let P be well typed in $\gamma ; \Gamma \vdash P \triangleright \Xi$ and reduce such that $P \longrightarrow_c Q$.

- If c is internal, then $\gamma ; \Gamma \vdash Q \triangleright \Xi$.
- If c is external i and $\Gamma \ni_i \ell_\# \triangleright \Delta$, then $\gamma ; \Delta \vdash Q \triangleright \Xi$.

Proof (Sketch). By induction on $P \longrightarrow_c Q$.

- Case comm: we apply framing (Thm. 1) (to rearrange the assumptions), substitution (Thm. 7) and strengthening (Thm. 3).
- Case par: by induction on the process that is being reduced.
- Case res: case split on channel c: if internal proceed inductively; if external 0 (i.e. the channel introduced by scope restriction) use Lem. 5 to subtract $\ell_\#$ from the channel's usage annotation and proceed inductively; if external $(1{+}i)$ proceed inductively.
- Case struct: we apply subject congruence (Thm. 5) and proceed inductively. ◀

6 Conclusion, Related and Future Work

Allais [1] uses leftover typing to mechanise in Agda a bidirectional type system for the linear λ-calculus. He proves type preservation and provides a decision procedure for type checking and type inference. In our paper, we apply leftover typing from Allais [1] to the π-calculus and focus on type checking. We give a more general usage algebra, leading to linear, graded and shared type systems. Drawing from quantitative type theory (by McBride and Atkey [20, 2]), in our work we are also able to talk about fully consumed resources. As such, we can transmit over a channel all none multiplicities of a fully exhausted channel, namely a channel of multiplicity $\ell_\varnothing$.

Recent years have seen an increase in the efforts to mechanise resource-aware concurrency calculi. One of the earliest works is the mechanisation of the linear π-calculus in Isabelle/HOL by Gay [11]. Gay encodes the π-calculus with linear and shared types using de Bruijn indices, a reduction relation and a type system posterior to the syntax. However, in his work context splits are extrinsic, and the types of used multiplicities erased. We make context splits intrinsic, preserve the ability to talk about used resources, and adopt a more general usage algebra: our framework integrates three different type systems into one.

The work by Goto et al. [14] is, to the best of our knowledge, the first formalisation of session types that includes a type safety proof (in Coq). The authors extend session types with polymorphism and pattern matching. They use a locally-nameless encoding for variable references, a syntax prior to types, and an LTS semantics that encodes session-typed processes into the π-calculus. Their type system uses reordering of contexts and extrinsic context splits. Castro et al. provide tooling for locally-nameless representations of process calculi in Coq [5] — in Coq de Bruijn indices are not as popular as in Agda or Idris because dependent pattern matching is far from straightforward. As a use-case, they use their tool to help automate proofs of subject reduction for a type system with session types. Taking an intrinsic approach to typing, Thiemann formalises in Agda the MicroSession (minimal GV [12]) calculus with support for recursion and subtyping [30]. Context splits are still given extrinsically, and exhausted resources are removed from typing contexts altogether. The semantics are given as an intrinsically typed CEK machine with a global context of session-typed channels. Most recently, Rouvoet et al. provide an intrinsic type system for a λ-calculus with session types [25]. They use proof relevant separation logic and a notion of a supply and demand *market* to make context splits transparent to the user. Their separation logic is based on a partial commutative monoid that need not be functional nor cancellative. Their typing rules preserve the balance between supply and demand, and are extremely elegant. They distill their typing rules even further by modelling the supply and demand market as a state monad.

To conclude, in this paper we provide a well scoped syntax and a semantics for the π-calculus, extrinsically define a type system on top of the syntax capable of handling linear, graded and shared types under the same unified framework and show that reduction preserves the well-typedness of processes. We avoid the need for extrinsic context splits by defining a type system based on leftover typing [1], which is defined here for the first time for the π-calculus. As a result, theorems like framing, weakening and strengthening can now we stated for the linear π-calculus. Our work is fully mechanised in around 1500 lines of code in Agda [32].

As future work, we intend to prove further properties of our type system, such as that reduction preserves the balancing of channels. We intend to add support for products, sum types and recursion to both our syntax and our type system. Orthogonally, making our typing rules bidirectional would allow us to provide a decision procedure for type checking processes in a given set of algebras. Furthermore, it might also be worth identifying correspondences between our usage algebra and particular state machines. Finally, we will use our π-calculus with leftovers as an underlying framework on top of which we can implement session types and other advanced type theories.

References

1 Guillaume Allais. Typing with Leftovers - A mechanization of Intuitionistic Multiplicative-Additive Linear Logic. page 22 pages, 2018. doi:10.4230/lipics.types.2017.1.

2 Robert Atkey. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 56–65, Oxford United Kingdom, July 2018. ACM. https://dl.acm.org/doi/10.1145/3209108.3209189. doi:10.1145/3209108.3209189.

3 Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Math. Struct. Comput. Sci.*, 6(6):579–612, 1996.

4 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158093.

5 David Castro, Francisco Ferreira, and Nobuko Yoshida. EMTST: Engineering the Meta-theory of Session Types. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 278–285, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-45237-7_17.

6 Ornela Dardha. Recursive session types revisited. In Marco Carbone, editor, *BEAT*, volume 162 of *EPTCS*, pages 27–34, 2014. https://doi.org/10.4204/EPTCS.162.4. doi:10.4204/EPTCS.162.4.

7 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP*, 2012. doi:10.1145/2370776.2370794.

8 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017. https://doi.org/10.1016/j.ic.2017.06.002. doi:10.1016/j.ic.2017.06.002.

9 Nicolaas Govert de Bruijn. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

10 Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, January 1994. doi:10.1007/BF01211308.

11 Simon J. Gay. A Framework for the Formalisation of Pi Calculus Type Systems in Isabelle/HOL. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, pages 217–232. Springer Berlin Heidelberg, 2001.

12 Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, January 2010. https://www.cambridge.org/core/product/identifier/S0956796809990268/type/journal_article. doi:10.1017/S0956796809990268.

13 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, January 1987. https://doi.org/10.1016/0304-3975(87)90045-4. doi:10.1016/0304-3975(87)90045-4.

14 Matthew Goto, Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, and James Riely. An extensible approach to session polymorphism. *Mathematical Structures in Computer Science*, 26(3):465–509, March 2016. https://www.cambridge.org/core/product/identifier/S0960129514000231/type/journal_article. doi:10.1017/S0960129514000231.

15 Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *LNCS*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

16 Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, 1998. doi:10.1007/BFb0053567.

17 Naoki Kobayashi. Type systems for concurrent programs. http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf, 2007.

18 Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *POPL*, 1996. doi:10.1145/237721.237804.

19 Nicholas D. Matsakis and Felix S. Klock II. The rust language. In *HILT*, pages 103–104. ACM, 2014. doi:10.1145/2663171.2663188.

20 Conor McBride. I Got Plenty o' Nuttin'. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 207–233. Springer International Publishing, Cham, 2016. `doi:10.1007/978-3-319-30936-1_12`.

21 Robin Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, May 1999.

22 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Inf. Comput.*, 100(1), 1992. `doi:10.1016/0890-5401(92)90008-4`.

23 Luca Padovani. A simple library implementation of binary sessions. *Journal of Functional Programming*, 27, 2017. `doi:10.1017/S0956796816000289`.

24 Benjamin C. Pierce and David N. Turner. Pict: a programming language based on the pi-calculus. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 455–494. The MIT Press, 2000.

25 Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 284–298, New Orleans LA USA, January 2020. ACM. `https://dl.acm.org/doi/10.1145/3372885.3373818`. `doi:10.1145/3372885.3373818`.

26 Davide Sangiorgi and David Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

27 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In *ECOOP*, volume 74 of *LIPIcs*, pages 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. `https://doi.org/10.4230/LIPIcs.ECOOP.2017.24`. `doi:10.4230/LIPIcs.ECOOP.2017.24`.

28 Alceste Scalas and Nobuko Yoshida. Lightweight session programming in scala. In *ECOOP*, 2016. `doi:10.4230/LIPIcs.ECOOP.2016.21`.

29 Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE*, volume 817 of *LNCS*, pages 398–413. Springer, 1994. `doi:10.1007/3-540-58184-7_118`.

30 Peter Thiemann. Intrinsically-Typed Mechanized Semantics for Session Types. *arXiv:1908.02940 [cs]*, August 2019. `http://arxiv.org/abs/1908.02940`. `arXiv:1908.02940`.

31 Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, page 561. North-Holland, 1990.

32 Uma Zalakain and Ornela Dardha. Typing the Linear π-calculus – Formalisation in Agda. 2020. `https://github.com/umazalakain/typing-linear-pi`.

A Lemmas

▶ **Lemma 1.** For every variables i and j, if $i \not\equiv j$ then $\mathrm{UNUSED}_i(P\ [\ i \mapsto j\ ])$.

Proof. By structural induction on Process and Var. ◀

▶ **Lemma 2.** The function $\mathrm{lower}_i\ P\ uP$ has an inverse $\mathrm{lift}_i\ P$ that increments every Var greater than or equal to i, such that $\mathrm{lift}_i\ (\mathrm{lower}_i\ P\ uP) \equiv P$.

Proof. By structural induction on Process and Var. ◀

▶ **Lemma 3.** The function $\mathrm{swap}_i\ P$ is its own inverse: $\mathrm{swap}_i\ (\mathrm{swap}_i\ P) \equiv P$.

Proof. By structural induction on Process and Var. ◀

▶ **Lemma 4.** For all well-typed processes $\gamma\ ;\ \Gamma \vdash P \triangleright \Xi$, if the variable i is unused within P, then Γ at i is equivalent to Ξ at i.

Proof. By induction on Process and Var. ◀

▶ **Lemma 5.** Every input usage context Γ of a well-typed process $\gamma\ ;\ \Gamma \vdash P \triangleright \Delta$ that reduces by communicating on a channel external to it (that is, $P \longrightarrow_{\mathrm{external}\ i} Q$ for some Q) has a multiplicity of at least $\ell_\#$ at index i.

Proof. By induction on the reduction derivation $P \longrightarrow_{\mathrm{external\ i}} Q$. ◀

B Structural Congruence

Structural congruence is a congruent equivalence relation. As such, rewrites can happen anywhere inside a process, and are closed under reflexivity, symmetry and transitivity as shown by the first row of Fig. 12. The rest of the rules defines structural congruence under a context $\mathcal{C}[\cdot]$ [26], respectively restriction, composition, input and output.

$$\frac{}{\mathrm{REC} : \mathrm{SET}}\ \textsc{Rec} \qquad \frac{}{\mathrm{zero} : \mathrm{REC}} \qquad \frac{r : \mathrm{REC}}{\mathrm{one}\ r : \mathrm{REC}} \qquad \frac{r\ s : \mathrm{REC}}{\mathrm{two}\ r\ s : \mathrm{REC}}$$

$$\frac{P\ Q : \mathrm{PROCESS}_n \qquad r : \mathrm{REC}}{P =_r Q : \mathrm{SET}}\ \textsc{Equals} \qquad \frac{eq : P \equiv Q}{\mathrm{struct}\ eq : P =_{\mathrm{zero}} Q}$$

$$\frac{eq : P =_r P'}{\text{cong-scope}\ eq : \boldsymbol{\nu}\ P =_{\mathrm{one}\ r} \boldsymbol{\nu}\ P'} \qquad \frac{eq : P =_r P'}{\text{cong-comp}\ eq : P \parallel Q =_{\mathrm{one}\ r} P' \parallel Q}$$

$$\frac{eq : P =_r P'}{\text{cong-recv}\ eq : x\ ()\ P =_{\mathrm{one}\ r} x\ ()\ P'} \qquad \frac{eq : P =_r P'}{\text{cong-send}\ eq : x\ \langle\ y\ \rangle\ P =_{\mathrm{one}\ r} x\ \langle\ y\ \rangle\ P'}$$

$$\frac{}{\mathrm{refl} : P =_{\mathrm{zero}} P} \qquad \frac{eq : P =_r Q}{\mathrm{sym}\ eq : Q =_{\mathrm{one}\ r} P} \qquad \frac{eq_1 : P =_r Q \qquad eq_2 : Q =_s R}{\mathrm{trans}\ eq_1\ eq_2 : P =_{\mathrm{two}\ r\ s} R}$$

Figure 12 Structural rewriting rules lifted to a congruent equivalence relation indexed by a recursion tree.

In the transitivity rule, we must show that if P is structurally congruent to Q and Q is to R, and P is well-typed, then so is R. To do so, we need to proceed by induction and first get a proof of the well-typedness of Q, then use that to reach R. To show the typechecker that the doubly recursive call terminates we index the equivalence relation $=$ by a type REC that models the structure of the recursion.

Subject congruence states that applying structural congruence rules to a well typed process preserves its well-typedness.

▶ **Theorem 9** (Subject congruence)**.** If $P = Q$ and $\gamma ; \Gamma \vdash P \triangleright \Xi$, then $\gamma ; \Gamma \vdash Q \triangleright \Xi$.

Proof (Sketch). The proof is by induction on EQUALS $=$. Here we only consider the nontrivial base cases for struct and their symmetric variants. Full proof in [32]. We proceed by induction on STRUCTCONG $\equiv$:

- Case comp-assoc: trivial, as leftover typing is naturally associative.
- Case comp-sym for $P \parallel Q$: we use framing (Thm. 1) to shift the output context of P to the one of Q; and the input context of Q to the one of P.
- Case comp-end: trivial, as the typing rule for $\mathbb{0}$ has the same input and output contexts.
- Case scope-end: we show that the usage annotation of the newly created channel must be $\ell_\varnothing$, making the proof trivial. In the opposite direction, we instantiate the newly created channel to a type $\mathbb{1}$ and a usage annotation $\ell_\varnothing$..
- Case scope-ext for $\nu\ P \parallel Q$: we to show that P preserves the usage annotations of the unused variable (Lem. 4) and then use strengthening (Thm. 3). In the reverse direction, we use weakening (Thm. 2) on P and show that lowering and then lifting P results in P (Lem. 2).
- Case scope-comm: we use swapping (Thm. 4), and for the reverse direction swapping and Lem. 3 to show that swapping two elements in P twice leaves P unchanged. ◀

C Substitution

▶ **Theorem 10** (Generalised substitution)**.** Let process P be well-typed in $\gamma ; \Gamma_i \vdash P \triangleright \Psi_i$. The substituted variable i is capable of m in Γ_i, and capable of n in Ψ_i. Substitution will take these usages m and n away from i and transfer them to the variable j we are substituting for. In other words, there must exist some Γ, Ψ, Γ_j and Ψ_j such that:

- $\gamma ; \Gamma_i \ni_i t ; m \triangleright \Gamma$
- $\gamma ; \Gamma_j \ni_j t ; m \triangleright \Gamma$
- $\gamma ; \Psi_i \ni_i t ; n \triangleright \Psi$
- $\gamma ; \Psi_j \ni_j t ; n \triangleright \Psi$

Let Γ and Ψ be related such that $\Gamma \coloneqq \Delta \otimes \Psi$ for some Δ. Let Δ have a usage annotation $\ell_\varnothing$ at position i, so that all consumption from m to n must happen in P. Then substituting i with j in P will be well-typed in $\gamma ; \Gamma_j \vdash P\,[\,i \mapsto j\,] \triangleright \Psi_j$. Refer to Fig. 11a for a diagramatic representation.

Proof (Sketch). By induction on the derivation $\gamma ; \Gamma_i \vdash P \triangleright \Psi_i$.

- For constructor end we get $\Gamma_i \equiv \Psi_i$. From $\Delta_i \equiv \ell_\varnothing$ follows that $m \equiv n$. Therefore $\Gamma_j \equiv \Psi_j$ and end can be applied.
- For constructor chan we proceed inductively, wrapping arrows $\ni_i m$, $\ni_j m$, $\ni_i n$ and $\ni_j n$ with 1+.
- For constructor recv we must split Δ to proceed inductively on the continuation. Observe that given the arrow from Γ_i to Ψ_i and given that Δ is $\ell_\varnothing$ at index i, there must exist some δ such that $m \coloneqq \delta \cdot^2 n$. l

  - If the input is on the variable being substituted, we split m such that $m \coloneqq \ell_{\mathrm{i}} \cdot^2 l$ for some l, and construct an arrow $\Xi_i \ni_i l \triangleright \Gamma$ for the inductive call. Similarly, we construct for some Ξ_j the arrows $\Gamma_j \ni_j \ell_{\mathrm{i}} \triangleright \Xi_j$ as the new input channel, and $\Xi_j \ni_j l \triangleright \Gamma$ for the inductive call.
  - If the input is on a variable x other than the one being substituted, we construct the arrows $\Xi_i \ni_i m \triangleright \Theta$ (for the inductive call) and $\Gamma \ni_x \ell_{\mathrm{i}} \triangleright \Theta$ for some Θ. We then construct for some Ξ_j the arrows $\Gamma_j \ni_x \ell_{\mathrm{i}} \triangleright \Xi_j$ (the new output channel) and $Xi_j \ni_j m \triangleright \Theta$ (for the inductive call). Given there exists a composition of arrows from Ξ_i to Ψ, we conclude that Θ splits Δ such that $\Gamma \coloneqq \Delta_1 \otimes \Theta$ and $\Theta \coloneqq \Delta_2 \otimes \Psi$. As $\ell_\varnothing$ is a minimal element, then Δ_1 must be $\ell_\varnothing$ at index i, and so must Δ_2.
- send applies the ideas outlined for the recv constructor to both the VarRef doing the output, and the VarRef for the sent data.
- For comp we first find a δ, Θ, Δ_1 and Δ_2 such that $\Xi_i \ni_i \delta \triangleright \Theta$ and $\Gamma \coloneqq \Delta_1 \otimes \Theta$ and $\Theta \coloneqq \Delta_2 \otimes \Psi$. Given Δ is $\ell_\varnothing$ at index i, we conclude that Δ_1 and Δ_2 are too. Observe that $m \coloneqq \delta \cdot^2 \psi$, where ψ is the usage annotation at index i consumed by the subprocess P. We construct an arrow $\Xi_j \ni_j \delta \triangleright \Theta$, for some Ξ_j. We can now make two inductive calls (on the derivation of P and Q) and compose their results.

◀