π with leftovers: a mechanisation in Agda *

Uma Zalakain[1][0000−0002−3268−9338] and Ornela Dardha[2][0000−0001−9927−7875]

[1] University of Glasgow, Scotland u.zalakain.1@research.gla.ac.uk
[2] University of Glasgow, Scotland ornela.dardha@glasgow.ac.uk

Abstract. Linear type systems need to keep track of how programs use their resources. The standard approach is to use *context splits* specifying how resources are (disjointly) split across subterms. In this approach, context splits redundantly echo information which is already present within subterms. An alternative approach is to use *leftover typing* [24,2], where in addition to the usual (input) usage context, typing judgments have also an output usage context: the leftovers. In this approach, the leftovers of one typing derivation are fed as input to the next, threading through linear resources while avoiding context splits. We use leftover typing to define a type system for a resource-aware π-calculus [28,27], a process algebra used to model concurrent systems. Our type system is parametrised over a set of *usage algebras* [21,35] that are general enough to encompass *shared types* (free to reuse and discard), *graded types* (use exactly n number of times) and *linear types* (use exactly once). Linear types are important in the π-calculus: they ensure privacy and safety of communication and avoid race conditions, while graded and shared types allow for more flexible programming. We provide a framing theorem for our type system, generalise the weakening and strengthening theorems to include linear types, and prove subject reduction. Our formalisation is fully mechanised in about 1850 lines of Agda [37].

Keywords: Pi-calculus · Linear types · Leftover typing · Concurrency · Mechanisation · Agda

1 Introduction

The π-calculus [28,27] is a computational model for communication and concurrency that boils concurrent processing down to the sending and receiving of data over communication channels. Notably, it features channel mobility: channels themselves are first class values and can be sent and received. Kobayashi et al. [23] introduced a typed version of the π-calculus with linear channel types, where channels must be used *exactly* once. Linearity in the π-calculus guarantees privacy and safety of communication and avoids race conditions.

* This work is supported by the EU HORIZON 2020 MSCA RISE project 778233 "Behavioural Application Program Interfaces" (BehAPI).

More broadly, linearity allows for resource-aware programming and more *efficient* implementations [36], and it inspired unique types (as in Clean [4]), and ownership types (as in Rust [25]). A linear type system must keep track of what resources are used in which parts of the program, and guarantee that they are *neither duplicated nor discarded*. To do so, the standard approach is to use context splits: typing rules for terms with multiple subterms add an extra side condition specifying what resources to allocate to each of the subterms. The typing derivations for the subterms must then use the entirety of their allocated resources. A key observation here is that each subterm already *knows* about the resources it needs. *Context splits contain usage information that is already present in the subterms.* Moreover, the subterms cannot be typed until the context splits have been defined. On top of that, using binary context splits means that typing rules with n subterms require $n - 1$ context splits, which considerably clutters the type system.

An alternative approach is *leftover typing*, a technique used to formulate intuitionistic linear logic [24] and to mechanise the linear λ-calculus [2]. Leftover typing changes the shape of the typing judgments and includes a second *leftover* output context that contains the resources that were left unused by the term. As a result, typing rules *thread* the resources *through* subterms without needing context splits: each subterm uses the resources it needs, and leaves the rest for its siblings. The first subterm in this chain of resources immediately knows what resources it has available.

In this paper, we use leftover typing to define for the first time a resource-aware type system for the π-calculus, and we fully mechanise our work in Agda [37]. All previous work on mechanisation of linear process calculi uses context splits instead [16,19,17,34,8]. We will further highlight the benefits of leftover typing as opposed to context splits in contributions and the rest of the paper.

Below we present two alternative typing rules for parallel composition in the linear π-calculus: the one on the left uses context splits, while the one on the right does not, and uses leftover typing instead:

$$\frac{\Gamma \coloneqq \Delta \otimes \Xi \qquad \Delta \vdash P \qquad \Xi \vdash Q}{\Gamma \vdash P \parallel Q} \qquad\qquad \frac{\Gamma \vdash P \triangleright \Delta \qquad \Delta \vdash Q \triangleright \Xi}{\Gamma \vdash P \parallel Q \triangleright \Xi}$$

Contributions and Structure of the Paper

1. **Leftover typing for resource-aware π-calculus**. Our type system uses leftover typing to model the resource-aware π-calculus (§ 4.3) and satisfies subject reduction (Theorem 5). In addition to making context splits unnecessary, leftover typing allows for a *framing* theorem (Theorem 1) to be stated and is naturally associative, making type safety properties considerably easier to reason about (§ 5). Thanks to leftover typing, we can now state *weakening* (Theorem 2) and *strengthening* (Theorem 3) for the whole framework, not just the shared fragment. This give a uniform and complete presentation of all the meta-theory for the resource-aware π-calculus.
2. **Shared, graded and linear unified π-calculus**. We generalise resource counting to a set of usage algebras that can be mixed within the same type

system. We do not instantiate our type system to only work with linear resources, instead we present an algebra-agnostic type system, and admit a mix of user-defined *resource aware* algebras [21,35] (§ 4.1). Any *partial commutative monoid* that is *decidable*, *deterministic*, *cancellative* and has a *minimal element* is a valid such algebra. Multiple algebras can be mixed in the type system — usage contexts keep information about what algebra to use for each type (§ 4.2). In particular, this allows for type systems combining linear (use exactly once), graded (exact number of n times) and shared (free to reuse and discard) types under the same framework.

3. **Full mechanisation in Agda**. The formalisation of the π-calculus with leftover typing, from the syntax to the semantics and the type system, has been fully mechanised in Agda in about 1850 lines of code, and is publicly available at [37]. We have fully mechanised all meta-theory and the details of a proof of subject reduction can be found in Appendix B.

We use type level de Bruijn indices [12,15] to define a syntax of π-calculus processes that is *well scoped by construction*: every free variable is accounted for in the type of the process that uses it (§ 2). We then provide an operational semantics for the π-calculus, prior to any typing (§ 3). This operational semantics is defined as a reduction relation on processes. The reduction relation tracks at the type level the channel on which communication occurs. This information is later used to state the subject reduction theorem. The reduction relation is defined modulo *structural congruence* — a relation defined on processes that acts as a quotient type to remove unnecessary syntactic minutiae introduced by the syntax of the π-calculus. We then define an interface for resource-aware algebras (§ 4.1) and use it to parametrise a type system based on leftover typing (§ 4.3). Finally, we present the meta theoretical properties of our type system in § 5.

Notation Data type definitions ($\mathbb{N}$) use double inference lines and index-free synonyms (Nat) as rule names for ease of reference. Constructors (0 and $1+$) are used as inference rule names. We maintain a close correspondence between the definitions presented in this paper and our mechanised definitions in Agda: inference rules become type constructors, premises become argument types and conclusions return types. Universe levels and universe polymorphism are omitted for brevity — all our types are of type SET. Implicit arguments are mentioned in type definitions but omitted by constructors.

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

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

2 Syntax

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: they are not inherently well scoped, one has to deal with alpha-conversion, and inserting new variables into a context entails proving that their names differ from all other names in context. To overcome these challenges, we use de Bruijn indices [12], where a natural number n (aka *index*) is used to refer to the variable introduced n binders ago. That is, binders no longer introduce names; terms at different *depths* use different indices to refer to the same binding.

While de Bruijn indices are useful for mechanisation, they are not as readable as names. To overcome this difficulty and demonstrate the correspondence between a π-calculus that uses names and one that uses de Bruijn indices, we provide *conversion functions* in both directions and prove that they are inverses of each other up to α-conversion. Further details can be found in Appendix A.

Definition 1 (Var and Process). A variable reference occurring under n binders can refer to n distinct variables. We introduce the indexed family of types [15] 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 free 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.

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

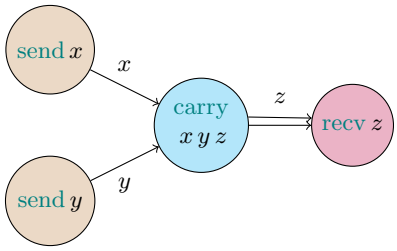
$$\frac{n : \mathbb{N}}{\mathrm{PROCESS}_n : \mathrm{SET}}\ \textsc{Process}$$

$$\begin{aligned}
\mathrm{PROCESS}_n ::= {} & \mathbb{0} && \text{(inaction)}\\
\mid {} & \boldsymbol{\nu}\, \mathrm{PROCESS}_{1+n} && \text{(restriction)}\\
\mid {} & \mathrm{PROCESS}_n \parallel \mathrm{PROCESS}_n && \text{(parallel)}\\
\mid {} & \mathrm{VAR}_n\, (\ )\, \mathrm{PROCESS}_{1+n} && \text{(input)}\\
\mid {} & \mathrm{VAR}_n \langle\, \mathrm{VAR}_n \,\rangle\, \mathrm{PROCESS}_n && \text{(output)}
\end{aligned}$$

Process $\mathbb{0}$ denotes the terminated process, where no further communications can occur; process $\boldsymbol{\nu}\, P$ creates a new channel and binds it at index 0 in the continuation process P; process $P \parallel Q$ composes P and Q in parallel; process $x\,(\ )\,P$ receives data along channel x and makes that data available at index 0 in the continuation process P; process $x \langle\, y \,\rangle\, P$ sends variable y over channel x and continues as process P.

Example 1 (The courier system).

We present a courier system that consists of three *roles*: a sender, who wants to send a package; a receiver, who receives the package sent by the sender; and a courier, who carries the package from the sender to the receiver.



Our courier system is defined by four π-calculus processes composed in parallel instantiating the above three roles: we have two sender processes, send x and send y, sending data over channels x and y, respectively; one receiver process, recv z, which receives over channel z the data sent from each of the senders – hence receives twice; and a courier process carry $x\,y\,z$, which synchronises communication among the senders and the receiver. The courier process first receives data from the two senders along its input channels x and y, and then sends the two received bits of data to the receiver along its output channel z.

The sender and receiver *roles* are defined below, parametrised by the channels on which they operate. The sender creates a new channel to be sent as data, and sends it over channel c, and then terminates. Processes send x and send y are an instantiation of send c. The receiver receives data *twice* on a channel c and then terminates. The receiver process recv z is an instantiation of recv c.

$$\text{send}\ c = \boldsymbol{\nu}\,(1{+}c\,\langle\,0\,\rangle\,\mathbb{0}) \qquad\qquad \text{recv}\ c\ = c\,(\ )\,(1{+}c)\,(\ )\,\mathbb{0}$$

The courier role is defined below as carry $x\ y\ z$. It sequentially receives on the two input channels x and y, instantiated as $in0$ and $in1$, and then outputs the two pieces of received data on the output channel z, instantiated as out. Finally, we create three communication channels and compose all four processes together: the first channel is shared between the one sender and the courier, the second between the other sender and the courier, and the third between the receiver and the courier. The result is the courier system defined below.

$$\begin{aligned}&\text{carry}\ in0\ in1\ out\ = in0\,(\ )\,(1{+}\ in1)\,(\ )\,(1{+}1{+}\ out)\,\langle\,1{+}0\,\rangle\,(1{+}1{+}\ out)\,\langle\,0\,\rangle\,\mathbb{0}\\&\text{system} = \boldsymbol{\nu}\,(\text{send}\ 0 \parallel \boldsymbol{\nu}\,(\text{send}\ 0 \parallel \boldsymbol{\nu}\,(\text{recv}\ 0 \parallel \text{carry}\ (1{+}1{+}0)\ (1{+}0)\ 0)))\end{aligned}$$

We continue this running example in § 4.3, where we provide typing derivations for the above processes and use a mix of linear, graded and shared typing to type the courier system.

3 Operational Semantics

Thanks to our well-scoped grammar in § 2, we now define the semantics of our language on the totality of the syntax.

Definition 2 (Unused). We consider a variable i to be unused in P ($\text{UNUSED}_i\ P$) if none of the inputs nor the outputs refer to it. $\text{UNUSED}_i\ P$ is defined as a

recursive predicate on P, incrementing i every time we go under a binder, and using $i \not\equiv x$ (which unfolds to the negation of propositional equality on VAR, i.e. $i \equiv x \to \bot$) to compare variables.

Definition 3 (STRUCTCONG). We define the base cases of a structural congruence relation $\cong$ as follows:

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

$$\frac{}{\text{comp-sym} : P \parallel Q \cong Q \parallel P} \qquad \frac{}{\text{comp-id} : P \parallel \mathbb{0}_n \cong P}$$

$$\frac{}{\text{scope-end} : \boldsymbol{\nu}\, \mathbb{0}_{1+n} \cong \mathbb{0}_n} \qquad \frac{uQ : \mathrm{UNUSED}_0\ Q}{\text{scope-ext} : \boldsymbol{\nu}\, (P \parallel Q) \cong (\boldsymbol{\nu}\, P) \parallel \mathrm{lower}_0\ Q\ uQ}$$

$$\frac{}{\text{scope-comm} : \boldsymbol{\nu}\, \boldsymbol{\nu}\, P \cong \boldsymbol{\nu}\, \boldsymbol{\nu}\, \mathrm{exchange}_0\ P}$$

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 side condition $\mathrm{UNUSED}_i\ Q$ makes sure that i is unused in Q (see Definition 2). The function $\mathrm{lower}_i\ Q\ uQ$ traverses Q decrementing every index greater than i. In scope-comm the function $\mathrm{exchange}_i\ P$ 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.

Definition 4 (EQUALS). We lift the relation STRUCTCONG $\cong$ and close it under equivalence and congruence in $\simeq$. This relation is structurally congruent under a context $\mathcal{C}[\cdot]$ [32] and is reflexive, symmetric and transitive.

Definition 5 (REDUCES). The operational semantics of the π-calculus is defined as a reduction relation $\longrightarrow_c$ indexed by the channel c on which communication occurs. We keep track of channel c so we can state subject reduction (Theorem 5).

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

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

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

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

$$\frac{eq_1 : P \simeq P' \qquad red : P' \longrightarrow_c Q' \qquad eq_2 : Q' \simeq Q}{\mathrm{struct}\ eq\ red : P \longrightarrow_c Q}$$

We distinguish between channels that are created inside 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 this substitution, $P\,[\,0 \mapsto {}_{1+}j\,]$ is *lowered* — all variable references are decreased by one (and we derive the 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 [32]. Rule res uses dec to decrement the index of channel c as we wrap processes P and Q inside a binder. It is defined as expected below:

$$\begin{aligned} \mathrm{dec}\ \mathrm{internal} &= \mathrm{internal} \\ \mathrm{dec}\ (\mathrm{external}\ 0) &= \mathrm{internal} \\ \mathrm{dec}\ (\mathrm{external}\ ({}_{1+}n)) &= \mathrm{external}\ n \end{aligned}$$

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 for multiplicities [21,35] that is satisfied by linear, graded and shared types alike. We then use pairs of multiplicities as usage annotations for a channel's input and output capabilities.

Definition 6 (Algebra). A *usage algebra* is a ternary relation $x \coloneqq y \cdot z$ that is *partial* (as not any two multiplicities can be combined), *deterministic* and *cancellative* (to aid equational reasoning) and *associative* and *commutative* (following directly from subject congruence for parallel composition). In addition, we ask that the leftovers can be *computed* so that we can automatically update the usage context every time input and output occurs — this is purely for usability. It has a *neutral element* $\cdot\text{-}0$ that is absorbed on either side, and that is also *minimal* (so that new resources cannot arbitrarily spring into life). It has an element $\cdot\text{-}1$ that is used to count inputs and outputs. Below we define such an algebra as a record 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 DEC P is a witness of either P or $P \to \bot$, where $\bot$ is the empty type with no constructors.)

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

We sketch the implementation of linear, graded and shared types as instances of our usage algebra below. Their use in typing derivations is illustrated in Example 3.

	carrier	operation
linear	$0 : \text{Lin}$ $1 : \text{Lin}$	$0 \coloneqq 0 \cdot 0$ $1 \coloneqq 1 \cdot 0$ $1 \coloneqq 0 \cdot 1$
graded	$0 : \text{Gra}$ $1+ : \text{Gra} \to \text{Gra}$	$\forall\, x\, y\, z$ $\to x \equiv y + z$ $\to x \coloneqq y \cdot z$
shared	$\omega : \text{Sha}$	$\omega \coloneqq \omega \cdot \omega$

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 7 (Algebras). An *indexed set of usage algebras* is a type IDX of indices that is nonempty (∃IDX) together with an interpretation USAGE of indices into types, and an interpretation ALGEBRAS of indices into usage algebras of the corresponding type.

$$\begin{aligned} &\mathrm{IDX} && : \mathrm{SET} \\ &\exists\mathrm{IDX} && : \mathrm{IDX} \\ &\mathrm{USAGE} && : \mathrm{IDX} \to \mathrm{SET} \\ &\mathrm{ALGEBRAS} && : (idx : \mathrm{IDX}) \to \mathrm{ALGEBRA}_{\mathrm{USAGE}_{idx}} \end{aligned}$$

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

Definition 8 (Type and PreCtx: types and typing contexts). A *type* is either a unit type (𝟙), or a channel type (C[t ; x]).

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

The unit type 𝟙 serves as a base case for types. The type C[t ; x] of a channel determines what type t of data and what usage annotations x are sent over that channel — we use the notation C^2 to stand for a C×C pair of input and output multiplicities, respectively. This channel notation aligns with $[t]$ $\mathbf{chan}_{(i^y, o^z)}$, where y, z are the input and output multiplicities, respectively [22]. Henceforth, we use $\ell_\emptyset$ to denote the multiplicity pair ·-0 , ·-0, ℓ_i for the pair ·-1 , ·-0, ℓ_o for ·-0 , ·-1, and $\ell_\#$ for ·-1 , ·-1. This notation was originally used in the linear π-calculus [23,32]. A *typing context* PRECTX_n is a length-indexed list of types that is either empty ([]) or the result of appending a type t : TYPE to an existing context (γ,t).

Definition 9 (Idxs and Ctx: contexts of indices and usage contexts). A context of indices IDXS_n is a length-indexed list that is either empty ([]) or the result of appending an index i : IDX to an existing context ($idxs$,i). A *usage context* is a context CTX_{idxs} indexed by a context of indices $idxs : \mathrm{IDXS}_n$ that is either empty ([]) or the result or appending a usage annotation pair u : USAGE^2_{idx} with index idx : IDX to an existing context (Γ,u).

4.3 Typing with Leftovers

We present a resource-aware type system for the π-calculus based on *leftover typing* [2], a technique 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 input to another typing derivation.

Leftover typing inverts the information flow of usage annotations so that it is the typing derivations of subprocesses which determine how resources are allocated. As a result, context split proofs are no longer necessary. Leftover typing also allows *framing* to be stated, and *weakening* and *strengthening* to cover linear types too.

Our type system is composed of two typing judgments: one for variable references (Definition 10) and one for processes (Definition 11). Both judgments 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 10 (VarRef: typing variable references). The VarRef typing relation for variable references is presented below.

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

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

We lift the operation $x \coloneqq y \cdot z$ and its algebraic properties to an operation $(x_l \,,\, x_r) \coloneqq (y_l \,,\, y_r) \cdot^2 (z_l \,,\, z_r)$ on pairs of multiplicities. The base case 0 splits the usage annotation x of type USAGE^2_{idx} into y and z (the leftovers). Note that the remaining context Γ is preserved unused as a leftover. This splitting $x \coloneqq y \cdot^2 z$ is as per the usage algebra provided by the developer for the index idx. In our Agda implementation, $x \coloneqq y \cdot^2 z$ is actually a trivially satisfiable implicit argument if $x \coloneqq y \cdot^2 z$ is inhabited and an unsatisfiable argument otherwise. 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 2 (Variable reference). egVar defines a variable reference $1{+}0$ with type $\mathrm{C}[\,\mathbb{1}\,;\ell_\mathrm{i}\,]$ and usage ℓ_i. We must show that this variable is well typed in an environment with a typing context $\gamma = [] \,,\, \mathrm{C}[\,\mathbb{1}\,;\ell_\mathrm{i}\,] \,,\, \mathbb{1}$ and a usage context $\Gamma = [] \,,\, \ell_\# \,,\, \ell_\#$. The VarRef constructors are completely determined by the variable index $1{+}\ 0$ in the type. 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 type $\mathrm{C}[\,\mathbb{1}\,;\ell_\mathrm{i}\,]$, and that the usage annotation $\ell_\#$ can be split such that $\ell_\# \coloneqq \ell_\mathrm{i} \cdot \ell_\mathrm{o}$ — using $\cdot$-compute$^\mathrm{r}$ to automatically fulfill the proof obligation.

```
egVar : ([] , C[ 𝟙 ; ℓᵢ ] , 𝟙) ; ([] , ℓ# , ℓ#) ∋₁₊₀ C[ 𝟙 ; ℓᵢ ] ; ℓᵢ ▷ ([] , ℓₒ , ℓ#)
egVar = 1+0
```

Definition 11 (Types: typing processes). The Types typing relation for the resource-aware π-calculus processes is presented below. For convenience, we reuse the constructor names introduced for the syntax in § 2.

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

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

$$\frac{\begin{array}{ll} chan : \gamma & ;\Gamma \quad \ni_i \mathrm{C}[\,t\,;x\,]\,;\ell_\mathrm{i} \triangleright \Xi \\ cont : \gamma\,,t\,;\Xi\,,x \vdash P & \triangleright \Theta\,,\ell_\emptyset \end{array}}{chan\,(\ )\,cont : \gamma\,;\Gamma \vdash i\,(\ )\,P \triangleright \Theta} \qquad \frac{\begin{array}{ll} chan : \gamma\,;\Gamma & \ni_i \mathrm{C}[\,t\,;x\,]\,;\ell_\mathrm{o} \triangleright \Delta \\ loc : \gamma\,;\Delta & \ni_j t\ \ ;x \triangleright \Xi \\ cont : \gamma\,;\Xi \vdash & P\ \ \triangleright \Theta \end{array}}{chan\,\langle loc \rangle\,cont : \gamma\,;\Gamma \vdash i\,\langle j \rangle\,P \triangleright \Theta}$$

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

The inaction process in rule $\mathbb{0}$ does not change usage annotations. The scope restriction in rule $\boldsymbol{\nu}$ 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 $(\ )$ requires a channel *chan* 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 determined by 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 $\langle\ \rangle$ requires a channel *chan* 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 transmitted data, which needs an element *loc* at index j with type t and usage x, as per the type of the channel *chan*. The leftovers Ξ are used to type the continuation process. Note that both indices i and j are determined by the syntax of the typed process. Parallel composition in rule $\parallel$ uses the leftovers of the left-hand process to type the right-hand process. Indeed, Theorem 4 shows that an alternative rule where the resources are first threaded through Q is admissible too.

Example 3 (Typing derivation (Continued)). We provide the typing derivation for the courier system defined in Example 1. For the sake of simplicity, we instantiate these processes with concrete variable references before typing them.

The receiver defined by the recv process receives data along the channel with index 0, which needs to be of type $\mathrm{C}[\,t\,;u\,]$ for some t and u. After receiving twice, the process ends: we must not be left with any unused multiplicities, thus $u = \ell_\emptyset$. We will use graded types to keep track of the exact number of times communication happens. Whatever the input multiplicity of the channel, we will consume 2 of it and leave the remaining as leftovers. The sender defined by the send process sends data along the channel with index 0, which needs to be of type $\mathrm{C}[\,t\,;u\,]$ for some t and u. We instantiate t (the type of data that the sender sends) to the trivial channel $\mathrm{C}[\,\mathbb{1}\,;\omega\,]$. As per the type of the process recv, $u = \ell_\emptyset$. We will transmit once, thus use $1{+}0$ output multiplicity, and leave the rest as leftovers. Agda can uniquely determine the arguments required by the $\boldsymbol{\nu}$ constructor.

```
recvwt : γ , C[ t ; ℓ∅ ] ; Γ , (1+1+l , r) ⊢ recv 0 ▷ Γ , (l , r)
recvwt = 0 ( ) (1+0) ( ) 𝟘
sendwt : γ , C[ C[ 𝟙 ; ω ] ; ℓ∅ ] ; Γ , (l , 1+r) ⊢ send 0 ▷ Γ , (l , r)
sendwt = ν _ _ ·-0 (1+0 ⟨ 0 ⟩ 𝟘)
```

Dually, the courier defined by the carry process expects input multiplicities for the channels shared with send and output multiplicities for the channel shared with recv. We can now compose these processes in parallel and type the courier system.

```
carrywt  : γ , C[ t ; ℓ∅ ] , C[ t ; ℓ∅ ] , C[ t ; ℓ∅ ]
         ; Γ , (1+lx , rx) , (1+ly , ry) , (lz , 1+1+rz)
         ⊢ carry (1+1+0) (1+0) 0
         ▷ Γ , (lx , rx) , (ly , ry) , (lz , rz)
carrywt  = (1+1+0) ( ) (1+1+0) ( ) (1+1+0) ⟨ 1+0 ⟩ (1+1+0) ⟨ 0 ⟩ 𝟘
systemwt : [] ; [] ⊢ system ▷ []
systemwt = ν _ _ _ (sendwt ∥ ν _ _ _ (sendwt ∥ ν _ _ _ (recvwt ∥ carrywt)))
```

5 Meta-Theory

We have mechanised subject reduction for our π-calculus with leftovers in 850 lines of Agda code. The meta-theory of resource-aware type systems often needs to reason on typing derivations modulo associativity in the allocation of resources. For type systems using context splitting side conditions, this means applying associativity lemmas to recompute context splits; for type systems using leftover typing it does not. As an example, the proof that comp-asssoc preserves typing proceeds by deconstructing the input derivation into $P \parallel (Q \parallel R)$ and reassembling it as $(P \parallel Q) \parallel R$ without the need of any extra reasoning.

All the reasoning carried out in our type safety proofs is based on the algebraic properties introduced in § 4.1 – the exception to this is ·-compute$^{\mathrm{r}}$, only there for the user's convenience. We lift the operation $x \coloneqq y \cdot^{2} z$ and its algebraic

properties to an operation $\Gamma \coloneqq \Delta \otimes \Xi$ on usage contexts that have the same underlying context of indices. The algebraic properties of the algebras allow us to see a typing derivation $\gamma \,;\, \Gamma \vdash P \triangleright \Delta$ as a unique *arrow* from Γ to Δ, and to freely compose and reason with arrows with the same typing context and a matching output and input usage contexts.

Leftover typing also allows us to state a *framing* theorem showing that adding or subtracting arbitrary usage annotations to the input and output usage contexts preserves typing – one can understand a typing derivation independently from its unused resources. With framing one can show that comp-comm preserves typing: in $P \parallel Q$ the typing of P and Q is independent of one another.

Theorem 1 (Framing). Let $\gamma \,;\, \Gamma_l \vdash P \triangleright \Xi_l$. Let Δ be such that $\Gamma_l \coloneqq \Delta \otimes \Xi_l$. Then for any Γ_r and Ξ_r where $\Gamma_r \coloneqq \Delta \otimes \Xi_r$ it holds that $\gamma \,;\, \Gamma_r \vdash P \triangleright \Xi_r$.

Leftover typing allows *weakening* and *strengthening* to acquire a more general form where linear variables can freely be added or removed from context too – as long as they are added and removed to and from both the input and output contexts.

Theorem 2 (Weakening). Let ins_i insert an element into a context at position i. 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 $\mathrm{ins}_i\ t\ \gamma \,;\, \mathrm{ins}_i\ x\ \Gamma \vdash \mathrm{lift}_i\ P \triangleright \mathrm{ins}_i\ x\ \Xi$.

Theorem 3 (Strengthening). Let del_i delete the element at position i from a context. Let P be well typed in $\gamma \,;\, \Gamma \vdash P \triangleright \Xi$. Let i be a variable not in P, such that $uP : \mathrm{UNUSED}_i\ P$. Then lowering every variable greater than i in P is well typed in $\mathrm{del}_i\ \gamma \,;\, \mathrm{del}_i\ \Gamma \vdash \mathrm{lower}_i\ P\ uP \triangleright \mathrm{del}_i\ \Xi$.

Subject congruence states that structural congruence (Definition 4) preserves the well-typedness of a process.

Theorem 4 (Subject Congruence). Let P and Q be processes. If $P \simeq Q$ and $\gamma \,;\, \Gamma \vdash P \triangleright \Xi$, then $\gamma \,;\, \Gamma \vdash Q \triangleright \Xi$.

Finally, subject reduction states that reducing on a channel c (Definition 5) preserves the well-typedness of a process — after consuming $\ell_\#$ from c if c is an external channel. Below we use $\Gamma \ni_i x \triangleright \Delta$ to stand for $\gamma \,;\, \Gamma \ni_i t \,;\, x \triangleright \Delta$ for some γ and t.

Theorem 5 (Subject Reduction). Let $\gamma \,;\, \Gamma \vdash P \triangleright \Xi$ and $P \longrightarrow_c Q$. If c is internal, then $\gamma \,;\, \Gamma \vdash Q \triangleright \Xi$. If c is external i and $\Gamma \ni_i \ell_\# \triangleright \Delta$, then $\gamma \,;\, \Delta \vdash Q \triangleright \Xi$.

We refer to Appendix B for a more detailed account of the mechanised proofs.

6 Conclusions, Related and Future Work

Extrinsic Encodings Extrinsic encodings define a syntax (often well-scoped) and a runtime semantics prior to any type system. This allows one to talk about

ill-typed terms, and defers the proof of subject reduction to a later stage. To the best of our knowledge, leftover typing makes its appearance in 1994, when Ian Mackie first uses it to formulate intuitionistic linear logic [24]. Allais [2] 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 this paper, we follow Allais [2] and apply leftover typing to the π-calculus for the first time. We generalise the usage algebra, leading to linear, graded and shared type systems. Drawing from quantitative type theory (by McBride and Atkey [26,3]), in our work we too are able to talk about fully consumed resources — e.g., we can transmit $\ell_\emptyset$ multiplicities of a fully exhausted channel. Recent years have seen an increase in the efforts to mechanise resource-aware process algebras, but one of the earliest works is the mechanisation of the linear π-calculus in Isabelle/HOL by Gay [16]. 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 typing rules demand user-provided context splits, and variables with consumed usage annotations are erased from context. We remove the demand for context splits, preserve the ability to talk about consumed resources, and adopt a more general usage algebra. Orchard et al. introduce Granule [29], a fully-fledged functional language with graded modal types, linear types, indexed types and polymorphism. Modalities include exact usages, security levels and intervals; resource algebras are pre-ordered semirings with partial addition. The authors provide bidirectional typing rules, and show the type safety of their semantics. The work by Goto et al. [19] is, to the best of our knowledge, the first formalisation of session types which comes along with a mechanised proof of type safety 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, which are not needed in our work.

Intrinsic Encodings Intrinsic encodings merge syntax and type system. As a result, one can only ever talk about well-typed terms, and the reduction relation by construction carries a proof of subject reduction. Significantly, by merging the syntax and static semantics of the object language one can fully use the expressive power of the host language. Thiemann formalises in Agda the MicroSession (minimal GV [17]) calculus with support for recursion and subtyping [34]. As Gay does in [16], context splits are given extrinsically, and exhausted resources are removed from typing contexts altogether. The runtime semantics are given as an intrinsically typed CEK machine with a global context of session-typed channels. In their recent paper, Ciccone and Padovani mechanise a dependently-typed linear π-calculus in Agda [8]. Their intrinsic encoding allows them to leverage Agda's dependent types to provide a dependently-typed interpretation of messages — to avoid linearity violations the interpretation of channel types is erased. Message input is modeled as a dependent function in Agda, and as a result message predicates, branching, and variable-length conversations can be

encoded. In contrast to our work, their algebra is on the multiplicities 0, 1, ω, and top-down context splitting proofs must be provided. In another recent work, Rouvoet et al. provide an intrinsic type system for a λ-calculus with session types [31]. 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 deterministic 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.

Other Work Castro et al. [6] provide tooling for locally-nameless representations of process calculi in Coq, where de Bruijn indices are less popular than in Agda or Idris. They use their tool to help automate proofs of subject reduction for a type system with session types. Orchard and Yoshida [30] embed a small effecftul imperative language into the session-typed π-calculus, showing that session types are expressive enough to encode effect systems. Based on contextual type theory, LINCX [18] extends the linear logical framework LLF [7] by internalising the notion of bindings and contexts. The result is a meta-theory in which HOAS encodings with both linear and dependent types can be described. The developer obtains for free an equational theory of substitution and decidable typechecking without having to encode context splits within the object language. Further work on mechanisation of the π-calculus [13,20,5,14,1], focuses on non-linear variations, differently from our range of linear, graded and shared types.

Conclusions and Future Work 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 subject reduction. We avoid extrinsic context splits by defining a type system based on leftover typing [2]. As a result, theorems like framing, weakening and strengthening can now be stated also for the linear π-calculus. Our work is fully mechanised in around 1850 lines of code in Agda [37].

As future work we intend to expand our framework to include infinite behaviour by adding process replication, which is challenging, as to prove subject congruence one needs to uniquely determine the resources consumed by a process — e.g., by adding type annotations to the syntax. Orthogonally, we aim to investigate making our typing rules bidirectional which would allow us to provide a decision procedure for type checking processes in a given set of algebras. Finally, we will use our π-calculus with leftovers as an underlying framework on top of which we can implement session types, via their encodings into linear types [9,11,33] and other advanced type theories.

Acknowledgments We want to thank Erika, Wen Kokke, James Wood, Guillaume Allais, Bob Atkey, and Conor McBride for their valuable suggestions.

References

1. Affeldt, R., Kobayashi, N.: A Coq Library for Verification of Concurrent Programs. Electron. Notes Theor. Comput. Sci. **199**, 17–32 (2008). https://doi.org/10.1016/j.entcs.2007.11.010
2. Allais, G.: Typing with Leftovers - A mechanization of Intuitionistic Multiplicative-Additive Linear Logic. In: Types for Proofs and Programs, TYPES. LIPIcs, vol. 104, pp. 1:1–1:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.TYPES.2017.1
3. Atkey, R.: Syntax and Semantics of Quantitative Type Theory. In: Logic in Computer Science, LICS. pp. 56–65. ACM (2018). https://doi.org/10.1145/3209108.3209189
4. Barendsen, E., Smetsers, S.: Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. Math. Struct. Comput. Sci. **6**(6), 579–612 (1996)
5. Bengtson, J.: The pi-calculus in nominal logic, vol. 2012 (2012), `https://www.isa-afp.org/entries/Pi_Calculus.shtml`
6. Castro, D., Ferreira, F., Yoshida, N.: EMTST: Engineering the Meta-theory of Session Types. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS. Lecture Notes in Computer Science, vol. 12079, pp. 278–285. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_17
7. Cervesato, I., Pfenning, F.: A Linear Logical Framework. In: Logic in Computer Science, LICS. pp. 264–275. IEEE Computer Society (1996). https://doi.org/10.1109/LICS.1996.561339
8. Ciccone, L., Padovani, L.: A Dependently Typed Linear π-Calculus in Agda. In: PPDP '20: 22nd International Symposium on Principles and Practice of Declarative Programming. pp. 8:1–8:14. ACM (2020). https://doi.org/10.1145/3414080.3414109
9. Dardha, O.: Recursive Session Types Revisited. In: Carbone, M. (ed.) Workshop on Behavioural Types, BEAT. EPTCS, vol. 162, pp. 27–34 (2014). https://doi.org/10.4204/EPTCS.162.4
10. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. In: Principles and Practice of Declarative Programming, PPDP. pp. 139–150. ACM (2012). https://doi.org/10.1145/2370776.2370794
11. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. Inf. Comput. **256**, 253–286 (2017). https://doi.org/10.1016/j.ic.2017.06.002, extended version of [10]
12. de Bruijn, N.G.: Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. In: Indagationes Mathematicae (Proceedings). vol. 75, pp. 381–392. Elsevier (1972)
13. Deransart, P., Smaus, J.: Subject Reduction of Logic Programs as Proof-Theoretic Property, vol. 2002 (2002), `http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/2002/S02-01/JFLP-A02-02.pdf`
14. Despeyroux, J.: A Higher-Order Specification of the pi-Calculus, Lecture Notes in Computer Science, vol. 1872. Springer (2000). https://doi.org/10.1007/3-540-44929-9_30
15. Dybjer, P.: Inductive Families. Formal Asp. Comput. **6**(4), 440–465 (1994). https://doi.org/10.1007/BF01211308
16. Gay, S.J.: A Framework for the Formalisation of Pi Calculus Type Systems in Isabelle/HOL. In: Theorem Proving in Higher Order Logics, TPHOLs. Lecture Notes in Computer Science, vol. 2152, pp. 217–232. Springer (2001). https://doi.org/10.1007/3-540-44755-5_16

17. Gay, S.J., Vasconcelos, V.T.: Linear type theory for asynchronous session types. J. Funct. Program. **20**(1), 19–50 (2010). https://doi.org/10.1017/S0956796809990268
18. Georges, A.L., Murawska, A., Otis, S., Pientka, B.: LINCX: A Linear Logical Framework with First-Class Contexts. In: European Symposium on Programming, ESOP, Lecture Notes in Computer Science, vol. 10201, pp. 530–555. Springer (2017). https://doi.org/10.1007/978-3-662-54434-1_20
19. Goto, M.A., Jagadeesan, R., Jeffrey, A., Pitcher, C., Riely, J.: An extensible approach to session polymorphism. Math. Struct. Comput. Sci. **26**(3), 465–509 (2016). https://doi.org/10.1017/S0960129514000231
20. Honsell, F., Miculan, M., Scagnetto, I.: pi-calculus in (Co)inductive-type theory. Theor. Comput. Sci. **253**(2), 239–285 (2001). https://doi.org/10.1016/S0304-3975(00)00095-5
21. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In: Rajamani, S.K., Walker, D. (eds.) Symposium on Principles of Programming Languages, POPL 2015. pp. 637–650. ACM (2015). https://doi.org/10.1145/2676726.2676980
22. Kobayashi, N.: Type Systems for Concurrent Programs (2007), `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf`
23. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the Pi-Calculus. In: Symposium on Principles of Programming Languages, POPL. pp. 358–371. ACM Press (1996). https://doi.org/10.1145/237721.237804
24. Mackie, I.: Lilac: A Functional Programming Language Based on Linear Logic. J. Funct. Program. **4**(4), 395–433 (1994). https://doi.org/10.1017/S0956796800001131
25. Matsakis, N.D., II, F.S.K.: The rust language. In: High integrity language technology, HILT. pp. 103–104. ACM (2014). https://doi.org/10.1145/2663171.2663188
26. McBride, C.: I Got Plenty o' Nuttin'. In: A List of Successes That Can Change the World, Lecture Notes in Computer Science, vol. 9600, pp. 207–233. Springer (2016). https://doi.org/10.1007/978-3-319-30936-1_12
27. Milner, R.: Communicating and mobile systems - the Pi-calculus. Cambridge University Press (1999)
28. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, Parts I and II. Inf. Comput. **100**(1) (1992). https://doi.org/10.1016/0890-5401(92)90008-4
29. Orchard, D., Liepelt, V., III, H.E.: Quantitative program reasoning with graded modal types. Proc. ACM Program. Lang. **3**(ICFP), 110:1–110:30 (2019). https://doi.org/10.1145/3341714
30. Orchard, D.A., Yoshida, N.: Using session types as an effect system. In: Gay, S., Alglave, J. (eds.) Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2015. EPTCS, vol. 203, pp. 1–13 (2015). https://doi.org/10.4204/EPTCS.203.1
31. Rouvoet, A., Poulsen, C.B., Krebbers, R., Visser, E.: Intrinsically-typed definitional interpreters for linear, session-typed languages. In: Certified Programs and Proofs, CPP. pp. 284–298. ACM (2020). https://doi.org/10.1145/3372885.3373818
32. Sangiorgi, D., Walker, D.: The Pi-Calculus - a theory of mobile processes. Cambridge University Press (2001)
33. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In: European Conference on Object-Oriented Programming, ECOOP. LIPIcs, vol. 74, pp. 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.ECOOP.2017.24

34. Thiemann, P.: Intrinsically-Typed Mechanized Semantics for Session Types pp. 19:1–19:15 (2019). https://doi.org/10.1145/3354166.3354184
35. Turon, A.J., Thamsborg, J., Ahmed, A., Birkedal, L., Dreyer, D.: Logical relations for fine-grained concurrency. In: Giacobazzi, R., Cousot, R. (eds.) Symposium on Principles of Programming Languages, POPL '13. pp. 343–356. ACM (2013). https://doi.org/10.1145/2429069.2429111
36. Wadler, P.: Linear Types can Change the World! In: Programming concepts and methods. p. 561. North-Holland (1990)
37. Zalakain, U., Dardha, O.: Typing the Linear π-Calculus – Formalisation in Agda (2021), `https://github.com/umazalakain/typing-linear-pi`

A From names to de Bruijn indices and back

The syntax of the π-calculus [32] using channel names is given by the RAW grammar below:

$$\frac{}{\text{RAW} : \text{SET}}\ \textsc{Raw}$$

$$\begin{array}{rll} \text{RAW} ::= & \mathbb{0} & \text{(inaction)} \\ \mid & (\boldsymbol{\nu}\ \text{NAME})\ \text{RAW} & \text{(restriction)} \\ \mid & \text{RAW} \parallel \text{RAW} & \text{(parallel)} \\ \mid & \text{NAME}\ (\ \text{NAME}\ )\ \text{RAW} & \text{(input)} \\ \mid & \text{NAME}\ \langle\ \text{NAME}\ \rangle\ \text{RAW} & \text{(output)} \end{array}$$

Channel names and variables range over x, y, z in NAME and processes over P, Q, R in RAW. Process $\mathbb{0}$ denotes the terminated process, where no further communications can occur. Process $(\boldsymbol{\nu}\, x)\ P$ 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 $(\boldsymbol{\nu}\, x)\ P$ and input $x\,(\,y\,)\ P$ are *binders*, they are the only constructs that introduce bound names — x and y in P, respectively.

In order to demonstrate the correspondence between a π-calculus that uses names and one that uses de Bruijn indices, we provide conversion functions in both directions and prove that they are inverses of each other up to α-conversion.

From names to de Bruijn indices When we translate into de Bruijn indices we keep the original binder names around — they will serve as name hints for when we translate back. The translation function fromRaw works recursively, keeping a context $ctx : \text{NAMES}_n$ that maps the first n indices to their names. Named references within the process are substituted with their corresponding de Bruijn index. We demand that the original process is well-scoped: that all its free variable names appear in ctx — this is decidable and we therefore automate the construction of such a proof term.

$$\begin{aligned} &\text{fromRaw} : (ctx : \text{NAMES}_n)\ (P : \text{RAW}) \\ &\qquad \to \text{WELLSCOPED}\ ctx\ P \to \text{PROCESS}_n \end{aligned}$$

From de Bruijn indices to names The translation function toRaw works recursively, keeping a context $ctx : \text{NAMES}_n$ that maps the first n indices to their names. As some widely-used languages do, this translation function produces unique variable names. These unique variable names use the naming scheme $< namehint >^{<n>}$, where $^{<n>}$ denotes that the name $< namehint >$ has already been bound n times before.

$$\text{toRaw} : (ctx : \text{NAMES}_n) \to \text{PROCESS}_n \to \text{RAW}$$

*Example 4 (*fromRaw *and* toRaw*).* We illustrate the conversion functions from names to de Bruijn indices (fromRaw) and back (toRaw) with three processes P, Q, R below.

$$\begin{aligned} P &= (\boldsymbol{\nu}\, x)\ (x\,(\,x\,)\quad x\,\langle\,z\,\rangle\quad \mathbb{0} \parallel (\boldsymbol{\nu}\, y)\ (x\,\langle\,y\,\rangle\quad y\,(\,y\,)\quad \mathbb{0})) \\ Q &= \boldsymbol{\nu}\quad (0\,(\,)\quad 0\,\langle\,2\,\rangle\quad \mathbb{0} \parallel \boldsymbol{\nu}\quad (1\,\langle\,0\,\rangle\quad 0\,(\,)\quad \mathbb{0})) \\ R &= (\boldsymbol{\nu}\, x^0)(x^0\,(\,x^1\,)\,x^1\,\langle\,z^0\,\rangle\,\mathbb{0} \parallel (\boldsymbol{\nu}\, y^0)(x^0\,\langle\,y^0\,\rangle\,y^0\,(\,y^1\,)\,\mathbb{0})) \end{aligned}$$

Process P uses names x, y, z and is translated via the conversion function fromRaw into process Q, which uses de Bruijn indices. Process Q is then translated via toRaw into process R, which follows the *Barendregt convention*[3] and is α-equivalent to the original process P.

In the following we present the main results that our conversion functions satisfy.

Lemma 1. Translating from de Bruijn indices to names via toRaw results in a well-scoped process.

Lemma 2. Translating from de Bruijn indices to names via toRaw results in a process that follows the *Barendregt convention*.

Lemma 3. Translating from de Bruijn indices to names and back via fromRaw $\circ$ toRaw results in the same process modulo internal variable name hints.

Lemma 4. Translating from names to de Bruijn indices and back via toRaw $\circ$ fromRaw results in the same process modulo α-conversion.

Proof. All the above results are proved by induction on Process, Var (Definition 1) and Raw. Complete details can be found in our mechanisation in Agda in [37].

B Type Safety

Exchange This property states that the exchange of two variables preserves the well-typedness of a process. We extend exchange_i introduced in Definition 4 to exchange types in typing contexts and usage annotations in usage contexts.

Theorem 6 (Exchange). Let P be well typed in $\gamma ; \Gamma \vdash P \triangleright \Xi$. Then, $\text{exchange}_i\ \gamma ; \text{exchange}_i\ \Gamma \vdash \text{exchange}_i\ P \triangleright \text{exchange}_i\ \Xi$.

Proof. All the above theorems are proved by induction on Types and VarRef. For details, refer to our mechanisation in Agda [37].

Subject Congruence This property states that applying structural congruence (Definition 4) to a well-typed process preserves its well-typedness. To prove this result, we must first introduce lemmas that establish that certain syntactic manipulations can be inverted (Lemma 5, Lemma 6) and how unused variables relate to the preservation of leftovers (Lemma 7).

Lemma 5. The function $\text{lower}_i\ P\ uP$ has an inverse $\text{lift}_i\ P$ that increments every Var greater than or equal to i, such that $\text{lift}_i\ (\text{lower}_i\ P\ uP) \equiv P$.

Proof. By structural induction on Process and Var.

Lemma 6. The function $\text{exchange}_i\ P$ is its own inverse: $\text{exchange}_i\ (\text{exchange}_i\ P) \equiv P$.

Proof. By structural induction on Process and Var.

[3] The Barendregt variable convention states that all bound variables/names in a process are distinct among each other and from the free variables/names.

Lemma 7. For all well-typed processes $\gamma \,;\, \Gamma \vdash P \triangleright \Xi$, if the variable i is unused within P, then Γ at i is equal to Ξ at i.

Proof. By induction on Process and Var.

We are now in a position to prove subject congruence.

Theorem 7 (Subject congruence). If $P \simeq Q$ and $\gamma \,;\, \Gamma \vdash P \triangleright \Xi$, then $\gamma \,;\, \Gamma \vdash Q \triangleright \Xi$.

Proof. The proof is by induction on Equals $\simeq$. Here we only consider those cases that are not purely inductive: the base cases for struct and their symmetric variants. Full proof in [37]. We proceed by induction on StructCong $\cong$:

- Case comp-assoc: trivial, as leftover typing is naturally associative.
- Case comp-sym for $P \parallel Q$: we use framing (Theorem 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 $\ell_\emptyset$, making the proof trivial. In the opposite direction, we instantiate the newly created channel to a type $\mathbb{1}$ and a usage annotation $\ell_\emptyset$.
- Case scope-ext for $\boldsymbol{\nu}\,(P \parallel Q)$: we need to show that P preserves the usage annotations of the unused variable (Lemma 7) and then use strengthening (Theorem 3). In the reverse direction, we use weakening (Theorem 2) on P and show that lowering and then lifting P results in P (Lemma 5).
- Case scope-comm: we use exchange (Theorem 6), and for the reverse direction exchange and Lemma 6 to show that exchanging two elements in P twice leaves P unchanged. □

Substitution This result is key to proving subject reduction. In Theorem 8 we prove a generalised version of substitution, where the substitition $P\,[\,i \mapsto j\,]$ is on any variable i. Then, in Theorem 9 we instantiate the generalised version to the concrete case where i is the most recently introduced variable 0, as required by subject reduction.

Theorem 8 (Generalised substitution). Let process P be well-typed in $\gamma \,;\, \Gamma_i \vdash P \triangleright \Psi_i$. The substituted variable at position i can be split into m in Γ_i, and into 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, let there be some Γ, Ψ, Γ_j and Ψ_j such that:

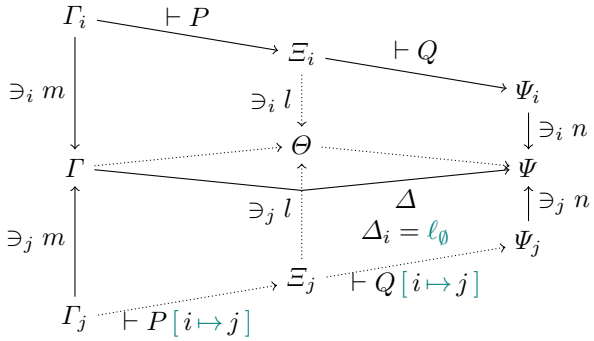
- $\gamma \,;\, \Gamma_i \ni_i t \,;\, m \triangleright \Gamma$
- $\gamma \,;\, \Gamma_j \ni_j t \,;\, m \triangleright \Gamma$
- $\gamma \,;\, \Psi_i \ni_i t \,;\, n \triangleright \Psi$
- $\gamma \,;\, \Psi_j \ni_j t \,;\, n \triangleright \Psi$

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

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

- For constructor $\mathbb{0}$ we get $\Gamma_i \equiv \Psi_i$. From $\Delta_i \equiv \ell_\emptyset$ follows that $m \equiv n$. Therefore $\Gamma_j \equiv \Psi_j$ and end can be applied.
- For constructor $\boldsymbol{\nu}$ we proceed inductively, wrapping arrows $\ni_i m$, $\ni_j m$, $\ni_i n$ and $\ni_j n$ with $1+$.

- For constructor $(\ )$ 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 $\ell_\emptyset$ at index i, there must exist some δ such that $m \coloneqq \delta \cdot^2 n.\,1$
  - If the input is on the variable being substituted, we split m such that $m \coloneqq \ell_i \cdot^2 l$ 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 $\Gamma_j \ni_j \ell_i \triangleright \Xi_j$ 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 $\Gamma \ni_x \ell_i \triangleright \Theta$ for some Θ. We then construct for some Ξ_j the arrows $\Gamma_j \ni_x \ell_i \triangleright \Xi_j$ (the new output channel) and $X i_j \ni_j m \triangleright \Theta$ (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 $\ell_\emptyset$ is a minimal element, then Δ_1 must be $\ell_\emptyset$ at index i, and so must Δ_2.
- $\langle\ \rangle$ applies the ideas outlined for the $(\ )$ constructor to both the VarRef doing the output, and the VarRef for the sent data.
- For $\|$ 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 $\ell_\emptyset$ 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.



Diagrammatic representation of the $\|$ case for substitution. Continuous lines represent known facts, dotted lines proof obligations.

Theorem 9 (Substitution). Let process P be well typed in $\gamma\,, t\,; \Gamma\,, m \vdash P \triangleright \Psi\,, \ell_\emptyset$. 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. For $\gamma\,; \Gamma \ni_j t\,; m \triangleright \Theta$ and $\gamma\,, t\,; \Theta\,, m \vdash P \triangleright \Xi\,, \ell_\emptyset$ for some Θ, we use framing to derive them. Then, we use these to apply Theorem 8.

Subject Reduction Finally we are ready to present our main result, stating that if P is well typed and it reduces to Q, then Q is well typed. The relation between the typing contexts used to type P and Q will be explained in Theorem 10. 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). We first introduce an auxiliary lemma:

Lemma 8. Every input usage context Γ of a well-typed process $\gamma\,;\Gamma \vdash P \triangleright \Delta$ that reduces by communicating on a channel external (that is, $P \longrightarrow_{\text{external } i} Q$ for some Q) has a multiplicity of at least $\ell_\#$ at index i.

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

Theorem 10 (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. By induction on $P \longrightarrow_c Q$. For the full details refer to our mechanisation in Agda.

- Case comm: we apply framing (Theorem 1) (to rearrange the assumptions), substitution (Theorem 9) and strengthening (Theorem 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 Lemma 8 to subtract $\ell_\#$ from the channel's usage annotation and proceed inductively; if external $(1{+}i)$ proceed inductively.
- Case struct: we apply subject congruence (Theorem 7) and proceed inductively. □