π with leftovers: a mechanisation in Agda

Anonymous Author(s)

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 can be built, such as *session types* [22, 23, 42].

We present an untyped syntax and semantics for the π-calculus, on top of which we provide a resource-aware type system parametrised by a set of multiplicity algebras. This allows the user to mix *linear*, *graded* and *shared* types, under the same unified framework. We use leftover typing [2] to encode our typing rules, thus avoiding the ubiquitous context splitting proofs common when dealing with linear type systems. We provide a framing theorem, generalise the weakening and strengthening theorems, and use them 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.

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

1 Introduction

We live in a concurrent and distributed 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, build, and predict such interactive systems; as mathematicians, abstraction is our tool of choice to do so.

The π-calculus [30, 31] 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 class values, and can therefore be sent and received. The π-calculus is at the basis of the design and implementation of programming languages for concurrency, such as Pict [37] and more recently Go [1]. At the state-of-the-art, the π-calculus features a wide plethora of types [25]: basic types, linear types, types for liveness properties, such as deadlock freedom, livelock freedom or termination, and most notably, session types. These make the π-calculus a fully-fledged tool for modelling and verifying concurrent and distributed systems.

In the early '90s, *linear types* for (functional) programming languages [6, 44] were introduced, following J.-Y. Girard's development of linear logic [20]. Linear type systems guarantee that resources are used *exactly once*. Enforcing this "no duplicating" and "no discarding" of resources via linearity allows for resource-aware programming and more efficient implementations [44]. Later on, linearity also inspired unique types, as in Clean [4], and ownership types, as in Rust [28].

Linearity influenced the π-calculus as well. Kobayashi et al. [26] introduced a typed version of the π-calculus where the linear type system restricts the usage of channels to exactly once. Meanwhile, linearity acquired a different flavour with the rise of *session types* [22, 23, 42], a formalism used in communication-centric programming where linearity ensures that a channel is owned by exactly one communicating participant — the channel itself may be used multiple times, as specified by its session type. Linearity in session types is the key ingredient that guarantees communication safety and session fidelity.

In recent years, an encoding of session types into linear types [10–12] has pushed the linear π-calculus [26] into the spotlight again. 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 in practice too. The encoding has been used as a technique to implement session types in mainstream programming languages such as OCaml [34] and Scala [40, 41], thus allowing to use the target linear π-calculus as an underlying theoretical and practical framework on top of which session types can be defined and implemented.

In this paper we present *the first full mechanisation in Agda of a π-calculus with linear, graded and shared types, all under a common unified framework.*

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] https://golang.org

While linear types are useful for resource-aware programming, graded and shared types are useful for a more flexible and mainstream programming.

We define a specification for an algebra on *multiplicities* — how many times a channel can be used — and use it to apply *leftover typing* for the first time to the π-calculus, along the lines of Allais [2]. The user is able to choose any mix of algebras for the type system — as long as they comply with the specification. This allows for linear, graded and shared types to be mixed in the same type system. Ultimately, we can exploit this formalisation as a unified framework for type-safe distributed modelling and programming using a variety of type systems, and build on top of it more advanced types, theories and languages.

Example 1 (The courier system)**.** We present an example that highlights the ways in which linear, graded and shared types can be used to prevent errors in a concurrent and distributed system defined by π-calculus processes in parallel.

A courier system 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 (given in Figure 1) is defined by four π-calculus processes in parallel instantiating the above three roles: we have two sender processes, S_1 and S_2, sending data over channels x and y, respectively; one receiver process, R, which receives over channel z the data sent from each of the senders – hence receives twice; and a courier process C, which synchronises communication among S_1, S_2 and R. The courier process C first receives data from S_1 and S_2 on its input channels x and y, respectively, and then sends the received data to R along its output channel z.

We will now define the courier system as π-calculus processes. In our work, the syntax of π-calculus processes adopts *de Bruijn indices* [13] for channel names and variables (details in § 2). In short, for the purposes of this example: process $\nu\, P$ creates a new channel and makes it available at index 0 in the continuation process P; $P \parallel Q$ composes processes 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. Both variable references VAR_n and processes $\mathrm{PROCESS}_n$ are indexed by the number n of free variables they can refer to.

For the sake of clarity, we will define the four subprocesses in our courier system separately and parametrise them by the channels on which they operate. Each role sender, receiver and courier is defined as a function respectively, send, recv and carry.

The sender *role* is defined below. The sender creates a new channel to be sent as data, and sends it over channel c, and then terminates, denoted by process $\mathbb{0}$.

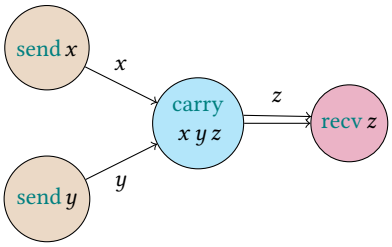

Figure 1. The courier system: two sender S_1 and S_2, one receiver R and one courier C.

$$\begin{aligned}&\mathrm{send} \;:\; \mathrm{VAR}_n \to \mathrm{PROCESS}_n\\ &\mathrm{send}\; c = \nu\, (1{+}c)\, \langle\, 0 \,\rangle\, \mathbb{0}\end{aligned}$$

Each sender *process* S_1 and S_2 is an instantiation of send c. In particular, S_1 is defined as send x and S_2 as send y, where data is sent over channels x and y, respectively (see Figure 1).

The receiver role is defined below, where recv c receives data *twice* on a channel c and then terminates with $\mathbb{0}$.

$$\begin{aligned}&\mathrm{recv} \;:\; \mathrm{VAR}_n \to \mathrm{PROCESS}_n\\ &\mathrm{recv}\; c \;= c\,(\ )\,(1{+}c)\,(\ \mathbb{0}\ )\end{aligned}$$

The receiver process R is defined as recv z, with channel z instantiating c, which is the channel used to receive data from courier (see Figure 1).

The courier role is defined below. The courier process C is defined as carry x y z. It sequentially receives on the two input channels x and y – instantiating $in0$ and $in1$, and then outputs the two pieces of received data on the output channel z – instantiating out (see Figure 1).

$$\begin{aligned}&\mathrm{carry} \;:\; \mathrm{VAR}_n \to \mathrm{VAR}_n \to \mathrm{VAR}_n \to \mathrm{PROCESS}_n\\ &\mathrm{carry}\; in0\; in1\; out \;=\\ &\qquad in0\,(\ )\\ &\qquad (1{+}\; in1)\,(\ )\\ &\qquad (1{+}1{+}\; out)\,\langle\, 1{+}0 \,\rangle\\ &\qquad (1{+}1{+}\; out)\,\langle\, 0 \,\rangle\, \mathbb{0}\end{aligned}$$

Finally, we compose these processes together and create three communication channels: the first channel is shared between the sender S_1 and the courier C, the second between the sender S_2 and the courier C, and the third between the receiver R and the courier C. The result is the courier systems defined below.

$$\begin{aligned}&\mathrm{system} \;:\; \mathrm{PROCESS}_0\\ &\mathrm{system} \;= \nu\,(\mathrm{send}\ 0\\ &\qquad \parallel \nu\,(\mathrm{send}\ 0\\ &\qquad \parallel \nu\,(\mathrm{recv}\ 0\\ &\qquad \parallel \mathrm{carry}\ (1{+}1{+}0)\ (1{+}0)\ 0)))\end{aligned}$$

In order to type the courier system we will use all three typing disciplines: linear, graded and shared typing. We will continue this example and provide the typing derivations in § 4.3. We use **linear types** for the *usage annotations* of data and **shared types** for the *data* itself that is being sent from the sender processes to the receiver. We use **graded types** for the usage annotations of the communication channels x, y, and z: the sender processes send x and send y each consume 1 output multiplicity and the receiver process recv z consumes 2 input multiplicities, as it receives twice on channel z. Dually, the courier process carry x y z consumes one input multiplicity on x, one input multiplicity on y, and two output multiplicities on z.

1.1 Contributions

1. **Formalisation of the syntax and the semantics of the π-calculus**:
   - **Syntax**: we use type level de Bruijn indices [13, 16] to introduce 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). For convenience and readability, the user can convert from and to a syntax using channel names (§ 2.1). These conversion functions are shown to be inverses of one another up to α-conversion.
   - **Semantics**: we provide an operational semantics for the π-calculus, prior to any typing (§ 3). Thanks to our well-scoped grammar (§ 2), the operational semantics is defined on *the totality of the syntax* as a reduction relation on processes (§ 3.2). 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 — aka type preservation (Theorem 8). The reduction relation is defined modulo *structural congruence* (§ 3.1) — a relation defined on processes that acts as a quotient type to remove unnecessary syntactic minutiae introduced by the syntax of the π-calculus.
2. **Leftover typing for the π-calculus with support for linear, graded and shared types**: we present our type system for a resource-aware π-calculus in § 4.
   - **Algebras on multiplicities**: The user can provide *resource-aware* algebras, which are then applied to the type system (§ 4.1). Any *partial commutative monoid* that is *decidable*, *deterministic*, *cancellative* and has a *minimal element* is a valid such algebra — linear types, graded types and shared types all satisfy these algebraic laws.
     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 under the same framework.
   - **Leftover typing**: 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 remaining as leftovers in the output usage context. Leftover typing **makes top-down context splits unnecessary**, allows for *framing* (Theorem 1) to be stated, and makes it possible to generalise *weakening* (Theorem 2) and *strengthening* (Theorem 3) for the π-calculus.
3. **Fully mechanised formalisation and type safety results**: 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.
   We present the type safety results of our π-calculus with leftovers in § 5. We have fully mechanised *framing* (Theorem 1), *weakening* (Theorem 2) and *strengthening* (Theorem 3). We use these results to then prove and mechanise *subject congruence* (Theorem 5), and together with *substitution* (Theorem 7), to prove and mechanise *subject reduction* (Theorem 8).

1.2 Notation

$$\frac{}{\mathbb{N} : \text{SET}}\ \text{NAT} \qquad \frac{}{0 : \mathbb{N}} \qquad \frac{n : \mathbb{N}}{1{+}n : \mathbb{N}}$$

Figure 2. Notation used in this paper.

Figure 2 illustrates the notation used in this paper. 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 given as inference rules. 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.

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

The syntax of the π-calculus [39] using channel names is given by the RAW grammar in Figure 3.

Channel names and variables range over x, y, z in NAME and processes over P, Q, R in RAW. Process 0 denotes the

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

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

Figure 3. Grammar with names.

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

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

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

Figure 4. Well-scoped grammar with type-level de Bruijn indices.

terminated process, where no further communications can 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 that 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: 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 [13], 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.

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 terms syntactically. In Figure 4 — which presents the syntax of the

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

Figure 5. From names to de Bruijn indices and back

π-calculus using de Bruijn indices — we do so by introducing the indexed family of types [16] 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.

2.1 From names to de Bruijn indices and back

In order to demonstrate the correspondence between an 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 $ctx : \mathrm{NAMES}_n$ 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.

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

From de Bruijn indices to names. The translation function toRaw works recursively, keeping a context $ctx : \mathrm{NAMES}_n$ 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 $< name - hint >^{<n>}$, where $^{<n>}$ denotes that the name $< name - hint >$ has already been bound n times before.

$$\mathrm{toRaw} : (ctx : \mathrm{NAMES}_n) \rightarrow \mathrm{PROCESS}_n \rightarrow \mathrm{RAW}$$

Example 2 (fromRaw and toRaw)**.** We illustrate the conversion functions from names to de Bruijn indices (fromRaw) and back (toRaw) with three processes P, Q, R in Figure 5.

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*[2] 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 ∘ toRaw results in the same process modulo internal variable name hints.

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

Proof. All the above results are proved by induction on Process and Var (Figure 4). Complete details can be found in our mechanisation in Agda. □

3 Operational Semantics

Thanks to our well-scoped grammar in Figure 4, the semantics of our language can now be defined on the totality of the syntax. We define structural congruence in § 3.1 and provide a reduction relation in § 3.2.

3.1 Structural Congruence

We define the base cases of a structural congruence relation StructCong $\cong$ in Figure 6.

$$\frac{}{P \cong Q : \mathrm{SET}}\ \textsc{StructCong}$$

$$\frac{}{\text{comp-assoc} : P \parallel (Q \parallel R) \cong (P \parallel Q) \parallel R}$$

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

$$\frac{}{\text{scope-end} : \nu\, \mathbb{0}_{1+n} \cong \mathbb{0}_n}$$

$$\frac{uQ : \mathrm{UNUSED}_0\ Q}{\text{scope-ext} : \nu\,(P \parallel Q) \cong (\nu\, P) \parallel \mathrm{lower}_0\ \ Q\ uQ}$$

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

Figure 6. Base cases of structural congruence.

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

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 (see Appendix A). 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.

We lift the relation StrucCong $\cong$ and close it under equivalence and congruence in Equals $\simeq$ as shown in Figure 7. The cong-* rules define structural congruence under a context $C[\cdot]$ [39], respectively scope restriction, parallel composition, input and output. The remaining three rules close structural congruence under reflexivity, symmetry and transitivity, respectively. The transitivity rule makes it tricky to prove predicates by induction on Equals: Agda cannot immediately see that such doubly recursive calls terminate. In order to remedy this, we model the recursive call structure syntactically and index Equals by a type Rec.

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

$$\frac{r\ s : \mathrm{REC}}{\mathrm{two}\ r\ s : \mathrm{REC}} \qquad \frac{P\ Q : \mathrm{PROCESS}_n \qquad r : \mathrm{REC}}{P \simeq_r Q : \mathrm{SET}}\ \textsc{Equals}$$

$$\frac{eq : P \cong Q}{\text{struct}\ eq : P \simeq_{\mathrm{zero}} Q} \qquad \frac{eq : P \simeq_r P'}{\text{cong-scope}\ eq : \nu\, P \simeq_{\mathrm{one}\ r} \nu\, P'}$$

$$\frac{eq : P \simeq_r P'}{\text{cong-comp}\ eq : P \parallel Q \simeq_{\mathrm{one}\ r} P' \parallel Q}$$

$$\frac{eq : P \simeq_r P'}{\text{cong-recv}\ eq : x\,(\ )\,P \simeq_{\mathrm{one}\ r} x\,(\ )\,P'}$$

$$\frac{eq : P \simeq_r P'}{\text{cong-send}\ eq : x\,\langle\, y\,\rangle\,P \simeq_{\mathrm{one}\ r} x\,\langle\, y\,\rangle\,P'} \qquad \frac{}{\mathrm{refl} : P \simeq_{\mathrm{zero}} P}$$

$$\frac{eq : P \simeq_r Q}{\mathrm{sym}\ eq : Q \simeq_{\mathrm{one}\ r} P} \qquad \frac{eq_1 : P \simeq_r Q \qquad eq_2 : Q \simeq_s R}{\mathrm{trans}\ eq_1\ eq_2 : P \simeq_{\mathrm{two}\ r\ s} R}$$

Figure 7. Structural rewriting rules lifted to a congruent equivalence relation indexed by a recursion tree.

3.2 Reduction Relation

The operational semantics of the π-calculus is defined as a reduction relation Reduces $\longrightarrow_c$ indexed by the channel c on which communication occurs (Figure 8). We keep track of channel c so we can state subject reduction (Theorem 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 this substitution, $P[0 \mapsto 1{+}j]$ is *lowered* — all variable references are decreased by one (we apply Lemma 5 to obtain a proof $\text{UNUSED}_0\ (P[0 \mapsto 1{+}j])$). 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 [39].

Lemma 5. For every variables i and j, if $i \not\equiv j$ then $\text{UNUSED}_i\ (P[i \mapsto j])$.

Proof. By structural induction on Process and Var. □

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

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

$$\frac{i : \text{VAR}_n}{\text{external}\ i : \text{CHANNEL}_n}$$

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

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

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

$$\frac{eq : P \simeq P' \qquad red : P' \longrightarrow_c Q}{\text{struct}\ eq\ red : P \longrightarrow_c Q}$$

Figure 8. 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 for multiplicities 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 1 (Usage algebra)**.** A *usage algebra* is a ternary relation $x := y \cdot z$ that is *partial* (as not any two multiplicities can be combined), *deterministic* and *cancellative* (to aid equational reasoning), *associative* and *commutative* (following directly from subject congruence for parallel composition), and in which the leftovers can be *computed* (as to automatically update the usage context every time input and output occurs). It has a *neutral element* $0\cdot$ that is absorbed on either side, and that is also *minimal* (so that new resources cannot arbitrarily spring into life). It has an element $1\cdot$ that is used to count inputs and outputs. Figure 9 defines such an algebra as a record ALGEBRA_C on a carrier C.

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

Figure 9. Usage algebra 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.

Figure 10 sketches the implementation of linear, graded and shared types as instances of our usage algebra. Their use in typing derivations is illustrated in Example 1 (Continued).

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

Figure 10. Example instances of USAGE and a ternary relation that satisfies our ALGEBRA interface.

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 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 (Figure 11).

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

Figure 11. Indexed set of usage algebras.

We keep typing contexts (PRECTX, in Figure 12) and usage contexts (CTX in Figure 13) 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]) (first two rows of Figure 12).

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 into the latter — e.g., 0 as booleans, 1 as natural numbers) so that we avoid having to deal with universe polymorphism. The type C[t ; x] of a channel determines what type t of data and what usage annotations x are sent over that channel. This channel notation aligns with $[t]$ $\mathbf{chan}_{(i^y, o^z)}$, where y, z are the multiplicities of input i and output o capabilities, respectively [25].

A *typing context* (last two rows of Figure 12) 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 pairs of usage annotations that is indexed by a length-indexed context of indices IDXS_n (Figure 13).

$$\frac{}{\mathrm{TYPE} : \mathrm{SET}}\ \textsc{Type} \qquad \frac{}{\mathbb{1} : \mathrm{TYPE}} \qquad \frac{n : \mathbb{N}}{\mathrm{B}[\, n \,] : \mathrm{TYPE}}$$

$$\frac{t : \mathrm{TYPE} \qquad idx : \mathrm{IDX} \qquad x : \mathrm{USAGE}^2_{idx}}{\mathrm{C}[\, t \,;\, x \,] : \mathrm{TYPE}}$$

$$\frac{n : \mathbb{N}}{\mathrm{PRECTX}_n : \mathrm{SET}}\ \textsc{PreCtx} \qquad \frac{}{[\,] : \mathrm{PRECTX}_0}$$

$$\frac{\gamma : \mathrm{PRECTX}_n \qquad t : \mathrm{TYPE}}{\gamma\,,\, t : \mathrm{PRECTX}_{1+n}}$$

Figure 12. Types and length-indexed typing contexts.

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

We use the notation C^2 to stand for a C×C pair of input and output multiplicities, respectively. Henceforth, we use $\ell_\varnothing$ 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 [26, 39].

$$\frac{n : \mathbb{N}}{\mathrm{IDXS}_n : \mathrm{SET}}\ \textsc{Idxs} \qquad \frac{}{[\,] : \mathrm{IDXS}_0}$$

$$\frac{idxs : \mathrm{IDXS}_n \qquad idx : \mathrm{IDX}}{idxs\,,\, idx : \mathrm{IDXS}_{1+n}} \qquad \frac{idxs : \mathrm{IDXS}_n}{\mathrm{CTX}_{idxs} : \mathrm{SET}}\ \textsc{Ctx}$$

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

Figure 13. Length-indexed context of carrier indices with a context of usage annotations on top.

4.3 Typing with Leftovers

We use *leftover typing* [2] 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.

Typing judgments for linear calculi typically carry a single context of usage annotations and thus the typing rules require frequent top-down proofs of context splitting — e.g.,

the typing rule for parallel composition would look like:

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

Input requires a similar context split; output requires two such context splits. This is rather cumbersome: for input and output, the proof of context split carries *exactly* the same information as the typing rule for variables; for parallel composition it means that the typing derivations of both subprocesses will have no context information whatsoever until the context split is determined.

Leftover typing solves this problem by inverting the flow of information so that **the typing derivations of the subprocesses determine the context split**. As a result, usage information is threaded bottom-up through the typing derivation, and top-down context split proofs are no longer needed. In addition, it allows *framing* (Theorem 1) to be stated, and *weakening* (Theorem 2) and *strengthening* (Theorem 3) to acquire a more general form.

Our type system with leftovers is composed of two typing relations: one for variable references (Definition 5) and one for processes (Definition 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 Figure 14.

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

$$\frac{x \coloneqq y \cdot^2 z}{0 : \gamma\,,t\,;\Gamma\,,x \ni_0 t\,;y \triangleright \Gamma\,,z}$$

$$\frac{loc_i : \gamma\,;\Gamma \ni_i t\,;x \triangleright \Delta}{1{+}\ loc_i : \gamma\,,t'\,;\Gamma\,,x' \ni_{1+i} t\,;x \triangleright \Delta\,,x'}$$

Figure 14. Typing rules for variable references.

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 and an operation $\Gamma \coloneqq \Delta \otimes \Xi$ on usage contexts with the same underlying context of indices.

The base case 0 splits the usage annotation x of type USAGE_{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. We use the *computability* 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 3 (Variable reference)**.** Listing 1 types a variable reference under a linear algebra.

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

```
1  _ : [] , C[ 𝟙 ; ℓᵢ ] , 𝟙 ; [] , ℓ# , ℓ# ∋₁₊₀ C[ 𝟙 ; ℓᵢ ] ; ℓᵢ ▷ [] , ℓₒ , ℓ#
2  _ = 1+0
3
4  _ : [] , C[ 𝟙 ; ℓᵢ ] ; [] , ℓ# ∋₀ C[ 𝟙 ; ℓᵢ ] ; ℓᵢ ▷ [] , ℓₒ
5  _ = 0
```

In Agda, the underscore _ introduces an anonymous declaration immediately followed by its definition. We must show that the variable at position $1{+}0$ is of type $\mathrm{C}[\,\mathbb{1}\,;\ell_\mathrm{i}\,]$ and has a usage annotation ℓ_i in an environment with a typing context $[]\,,\mathrm{C}[\,\mathbb{1}\,;\ell_\mathrm{i}\,]\,,\mathbb{1}$ and a usage context $[]\,,\ell_\#\,,\ell_\#$. The VarRef constructors we must use are completely determined by $1{+}\ 0$ in the type. The constructor $1{+}$ steps under the outermost variable in the context, preserving its usage annotation $\ell_\#$ from input to output, the result is the obligation on line 4. 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 $\ell_\# \coloneqq \ell_\mathrm{i} \cdot \ell_\mathrm{o}$ — we use $\cdot$-compute$^\mathrm{r}$ to fulfill this proof obligation automatically.

Definition 6 (Types: typing processes)**.** The Types typing relation for π-calculus processes is presented in Figure 15. For convenience, we reuse the constructor names introduced for the syntax in Figure 4.

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 $chan_i$ 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

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

$$\frac{}{\mathbb{0} : \gamma\,;\Gamma \vdash \mathbb{0} \triangleright \Gamma}$$

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

$$\frac{\begin{array}{ll} chan_i : \gamma & ;\Gamma \quad \ni_i \mathrm{C}[\,t\,;x\,]\,;\ell_i \triangleright \Xi \\ cont & : \gamma\,,t\,;\Xi\,,x \vdash P \quad \triangleright \Theta\,,\ell_\varnothing \end{array}}{chan_i\ (\ )\ cont : \gamma\,;\Gamma \vdash i\ (\ )\ P \triangleright \Theta}$$

$$\frac{\begin{array}{ll} chan_i : \gamma\,;\Gamma \ni_i \mathrm{C}[\,t\,;x\,]\,;\ell_o \triangleright \Delta & \\ loc_j & : \gamma\,;\Delta \ni_j t \quad ;x \triangleright \Xi \\ cont & : \gamma\,;\Xi \vdash P \quad \triangleright \Theta \end{array}}{chan_i\ \langle\ loc_j\ \rangle\ cont : \gamma\,;\Gamma \vdash i\ \langle\ j\ \rangle\ P \triangleright \Theta}$$

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

Figure 15. Leftover typing for a resource-aware type system.

in rule $\langle\ \rangle$ 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 transmitted data, 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 $\parallel$ 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 a top-down proof of context split.

Example 1 (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, thus that variable needs to be of channel type $\mathrm{C}[\,t\,;u\,]$ for some t and u. After receiving twice, the process ends, and we should not be left with any unused multiplicities: u needs to be $\ell_\varnothing$. 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. Once the types are given, the typing derivation is completely syntax directed.

$$\vdash \mathrm{recv}\ \gamma\,,\mathrm{C}[\,t\,;\ell_\varnothing\,]\,;\Gamma\,,(1{+}1{+}l\,,r) \vdash \mathrm{recv}\ 0 \triangleright \Gamma\,,(l\,,r)$$
$$\vdash \mathrm{recv}\ =\ 0\ (\ )\ (1{+}0)\ (\ )\ \mathbb{0}$$

The sender defined by the send process sends data along the channel with index 0, thus that variable needs to be of channel type $\mathrm{C}[\,t\,;u\,]$ for some t and u. We instantiate t (the type of data that the sender sends) to $\mathrm{C}[\,\mathbb{1}\,;\omega\,]$, as that is a value we can easily construct. As per the type of the process recv, the transmitted multiplicities u are $\ell_\varnothing$. We will transmit once, thus use a single output multiplicity, and leave the rest as leftovers. Agda can uniquely determine the arguments required by the ν constructor, and thus we can omit them using underscores.

$$\vdash \mathrm{send}\ \gamma\,,\mathrm{C}[\,\mathrm{C}[\,\mathbb{1}\,;\omega\,]\,;\ell_\varnothing\,]\,;\Gamma\,,(l\,,1{+}r) \vdash \mathrm{send}\ 0 \triangleright \Gamma\,,(l\,,r)$$
$$\vdash \mathrm{send}\ =\ \nu\ \_\ \_\ 0{\cdot}\ (1{+}0)\ \langle\ 0\ \rangle\ \mathbb{0}$$

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. The typing derivation here is uniquely determined by the type and syntax of the process.

$$\begin{array}{ll} \vdash \mathrm{carry} & : \gamma\,,\mathrm{C}[\,t\,;\ell_\varnothing\,]\,,\mathrm{C}[\,t\,;\ell_\varnothing\,]\,,\mathrm{C}[\,t\,;\ell_\varnothing\,] \\ & ;\Gamma\,,(1{+}lx\,,rx)\,,(1{+}ly\,,ry)\,,(lz\,,1{+}1{+}rz) \\ & \vdash \mathrm{carry}\ (1{+}1{+}0)\ (1{+}0)\ 0 \\ & \triangleright \Gamma\,,(lx\,,rx)\,,(ly\,,ry)\,,(lz\,,rz) \\ \vdash \mathrm{carry} & = \\ & (1{+}1{+}0)\ (\ ) \\ & (1{+}1{+}0)\ (\ ) \\ & (1{+}1{+}0)\ \langle\ 1{+}0\ \rangle \\ & (1{+}1{+}0)\ \langle\ 0\ \rangle\ \mathbb{0} \end{array}$$

We can now compose these processes in parallel and type the courier system. Specifying the types of the channels is not required: Agda can infer them from the types of the processes.

$$\begin{array}{ll} \vdash \mathrm{system} & :\ [\,]\,;[\,] \vdash \mathrm{system} \triangleright [\,] \\ \vdash \mathrm{system} & = \nu\ \_\ \_\ \_\ (\vdash \mathrm{send} \\ & \parallel \nu\ \_\ \_\ \_\ (\vdash \mathrm{send} \\ & \parallel \nu\ \_\ \_\ \_\ (\vdash \mathrm{recv} \\ & \parallel \vdash \mathrm{carry}))) \end{array}$$

5 Type Safety

5.1 Auxiliary Results

We will start this section with auxiliary results which are needed to prove two main theorems: subject congruence (Theorem 5) and subject reduction (Theorem 8) for our π-calculus with leftovers.

Framing. This property 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 $\Gamma_l := \Delta \otimes \Xi_l$. Let Γ_r and Ξ_r be arbitrary contexts such that $\Gamma_r := \Delta \otimes \Xi_r$. Then $\gamma ; \Gamma_r \vdash P \triangleright \Xi_r$.

Weakening. This property 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. This property 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 $uP \ : \ \mathrm{UNUSED}_i\ P$. Then lowering every variable greater than i in P is well typed in $\mathrm{del}_i\ \gamma ; \mathrm{del}_i\ \Gamma \vdash \mathrm{lower}_i\ P\ uP \triangleright \mathrm{del}_i\ \Xi$.

Exchange. This property states that the exchange of two variables preserves the well-typedness of a process. We extend $\mathrm{exchange}_i$ introduced in § 3.1 to exchange types in typing contexts and usage annotations in usage contexts.

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

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

5.2 Towards Subject Reduction

Subject Congruence. This property states that applying structural congruence (§ 3.1) 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 6, Lemma 7) and how unused variables relate to the preservation of leftovers (Lemma 8).

Lemma 6. 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 7. The function $\mathrm{exchange}_i\ P$ is its own inverse: $\mathrm{exchange}_i\ (\mathrm{exchange}_i\ P) \equiv P$.

Proof. By structural induction on Process and Var. □

Lemma 8. 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. □

We are now in a position to prove subject congruence.

Theorem 5 (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. 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 $\ell_\varnothing$, 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 (Lemma 8) 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 6).
- Case scope-comm: we use exchange (Theorem 4), and for the reverse direction exchange and Lemma 7 to show that exchanging two elements in P twice leaves P unchanged. □

Substitution. This result is key to proving subject reduction. In Theorem 6 we prove a generalised version of substitution, where the substitition $P\ [\ i \mapsto j\ ]$ is on any variable i. Then, in Theorem 7 we instantiate 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 $\Gamma := \Delta \otimes \Psi$ 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 to j in P will be well typed in $\gamma ; \Gamma_j \vdash P\ [\ i \mapsto j\ ] \triangleright \Psi_j$.

Proof. By induction on $\gamma \,;\, \Gamma_i \vdash P \triangleright \Psi_i$. Full proof can be found in Appendix B.

- Constructor $\mathbb{0}$: observe that $\Gamma_i \equiv \Psi_i$ and thus $\Gamma_j \equiv \Psi_j$.
- Constructor $\boldsymbol{\nu}$: we proceed inductively.
- Constructors $(\ )$, $\langle\ \rangle$ and $\|$: we must find Θ in Figure 16 and split the diagram along its vertical axis to proceed by induction. □

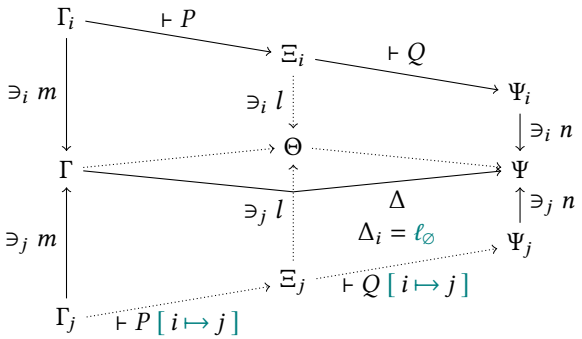

Figure 16. Diagrammatic representation of the $\|$ case for substitution. Continuous lines represent known facts, dotted lines proof obligations.

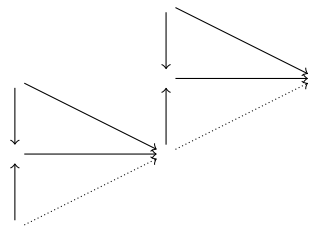

Figure 17. Alternative approach without arrows $\ni_i n$ and $\ni_j n$. This approach makes composing typing derivations before and after subsitution more difficult.

Theorem 7 (Substitution)**.** Let process 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. For $\gamma \,;\, \Gamma \ni_j t \,;\, m \triangleright \Theta$ and $\gamma, t \,;\, \Theta, m \vdash P \triangleright \Xi, \ell_\varnothing$ for some Θ, we use framing to derive them. Then, we use these to apply Theorem 6. □

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

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 8. 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). We first introduce an auxiliary lemma:

Lemma 9. 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 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. 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 7) 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 9 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 5) and proceed inductively. □

6 Related 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 [27]. 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 [3, 29]), in our work we too are able to talk about fully consumed resources — e.g., we can transmit $\ell_\varnothing$ 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 [17]. 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 [32], 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. [21] 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, whether they 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 [18]) calculus with support for recursion and subtyping [43]. As Gay does [17], 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 [9]. 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 [38]. 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. [7] provide tooling for locally-nameless representations of process calculi in Coq — 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.

Orchard and Yoshida [33] embed a small effectul imperative language into the session-typed π-calculus, showing that session types are expressive enough to encode effect systems.

Based on contextual type theory [35, 36], LINCX [19] extends the linear logical framework LLF [8] by internalising the notion of bindings and contexts. The result is a metatheory 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 mechanising the π-calculus [1, 5, 14, 15, 24], focuses on non-linear variations, whereas we present a range of linear, graded and shared types.

7 Conclusion and Future Work

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 [2], 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 1850 lines of code in Agda.

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] Reynald Affeldt and Naoki Kobayashi. A coq library for verification of concurrent programs. *Electron. Notes Theor. Comput. Sci.*, 199:17–32, 2008. doi:10.1016/j.entcs.2007.11.010.

[2] Guillaume Allais. Typing with leftovers - A mechanization of intuitionistic multiplicative-additive linear logic. In *Types for Proofs and Programs, TYPES*, volume 104 of *LIPIcs*, pages 1:1–1:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.TYPES.2017.1.

[3] Robert Atkey. Syntax and semantics of quantitative type theory. In *Logic in Computer Science, LICS*, pages 56–65. ACM, 2018. doi:10.1145/3209108.3209189.

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

[5] Jesper Bengtson. *The pi-calculus in nominal logic*, volume 2012. 2012. URL: https://www.isa-afp.org/entries/Pi_Calculus.shtml.

[6] 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):5:1–5:29, 2018. doi:10.1145/3158093.

[7] David Castro, Francisco Ferreira, and Nobuko Yoshida. EMTST: engineering the meta-theory of session types. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 12079 of *Lecture Notes in Computer Science*, pages 278–285. Springer, 2020. doi:10.1007/978-3-030-45237-7_17.

[8] Iliano Cervesato and Frank Pfenning. A linear logical framework. In *Logic in Computer Science, LICS*, pages 264–275. IEEE Computer Society, 1996. doi:10.1109/LICS.1996.561339.

[9] Luca Ciccone and Luca Padovani. A Dependently Typed Linear π-Calculus in Agda. In *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming (PPDP'20)*. ACM, 2020. To appear.

[10] Ornela Dardha. Recursive session types revisited. In Marco Carbone, editor, *Workshop on Behavioural Types, BEAT*, volume 162 of *EPTCS*, pages 27–34, 2014. doi:10.4204/EPTCS.162.4.

[11] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *Principles and Practice of Declarative Programming, PPDP*, pages 139–150. ACM, 2012. doi:10.1145/2370776.2370794.

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

[13] 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.

[14] Pierre Deransart and Jan-Georg Smaus. *Subject Reduction of Logic Programs as Proof-Theoretic Property*, volume 2002. 2002. URL: http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/2002/S02-01/JFLP-A02-02.pdf.

[15] Joëlle Despeyroux. *A Higher-Order Specification of the pi-Calculus*, volume 1872 of *Lecture Notes in Computer Science*. Springer, 2000. doi:10.1007/3-540-44929-9_30.

[16] Peter Dybjer. Inductive families. *Formal Asp. Comput.*, 6(4):440–465, 1994. doi:10.1007/BF01211308.

[17] Simon J. Gay. A framework for the formalisation of pi calculus type systems in isabelle/hol. In *Theorem Proving in Higher Order Logics, TPHOLs*, volume 2152 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2001. doi:10.1007/3-540-44755-5_16.

[18] Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010. doi:10.1017/S0956796809990268.

[19] Aina Linn Georges, Agata Murawska, Shawn Otis, and Brigitte Pientka. LINCX: A linear logical framework with first-class contexts. In *European Symposium on Programming, ESOP*, volume 10201 of *Lecture Notes in Computer Science*, pages 530–555. Springer, 2017. doi:10.1007/978-3-662-54434-1_20.

[20] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi:10.1016/0304-3975(87)90045-4.

[21] Matthew A. Goto, Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, and James Riely. An extensible approach to session polymorphism. *Math. Struct. Comput. Sci.*, 26(3):465–509, 2016. doi:10.1017/S0960129514000231.

[22] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *Conference on Concurrency Theory, CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

[23] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming ESOP*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

[24] Furio Honsell, Marino Miculan, and Ivan Scagnetto. pi-calculus in (co)inductive-type theory. *Theor. Comput. Sci.*, 253(2):239–285, 2001. doi:10.1016/S0304-3975(00)00095-5.

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

[26] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Symposium on Principles of Programming Languages, POPL*, pages 358–371. ACM Press, 1996. doi:10.1145/237721.237804.

[27] Ian Mackie. Lilac: A functional programming language based on linear logic. *J. Funct. Program.*, 4(4):395–433, 1994. doi:10.1017/S0956796800001131.

[28] Nicholas D. Matsakis and Felix S. Klock II. The rust language. In *High integrity language technology, HILT*, pages 103–104. ACM, 2014. doi:10.1145/2663171.2663188.

[29] Conor McBride. I got plenty o' nuttin'. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016. doi:10.1007/978-3-319-30936-1_12.

[30] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

[31] 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.

[32] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.*, 3(ICFP):110:1–110:30, 2019. doi:10.1145/3341714.

[33] Dominic A. Orchard and Nobuko Yoshida. Using session types as an effect system. In Simon Gay and Jade Alglave, editors, *Proceedings Eighth International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2015, London, UK, 18th April 2015*, volume 203 of *EPTCS*, pages 1–13, 2015. doi:10.4204/EPTCS.203.1.

[34] Luca Padovani. A simple library implementation of binary sessions. *J. Funct. Program.*, 27:e4, 2017. doi:10.1017/S0956796816000289.

[35] Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. pages 371–382, 2008. doi:10.1145/1328438.1328483.

[36] Brigitte Pientka, Andreas Abel, Francisco Ferreira, David Thibodeau, and Rébecca Zucchini. Cocon: Computation in contextual type theory. *CoRR*, abs/1901.03378, 2019. URL: http://arxiv.org/abs/1901.03378, arXiv:1901.03378.

[37] Benjamin C. Pierce and David N. Turner. Pict: a programming language based on the pi-calculus. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 455–494. The MIT Press, 2000.

[38] Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In *Certified Programs and Proofs, CPP*, pages 284–298. ACM, 2020. doi:10.1145/3372885.3373818.

[39] Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

[40] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In *European Conference on Object-Oriented Programming, ECOOP*, volume 74 of *LIPIcs*, pages 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ECOOP.2017.24.

[41] Alceste Scalas and Nobuko Yoshida. Lightweight session programming in scala. In *European Conference on Object-Oriented Programming, ECOOP*, volume 56 of *LIPIcs*, pages 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.21.

[42] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *Parallel Architectures and Languages Europe, PARLE*, volume 817 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 1994. doi:10.1007/3-540-58184-7_118.

[43] Peter Thiemann. Intrinsically-typed mechanized semantics for session types. pages 19:1–19:15, 2019. doi:10.1145/3354166.3354184.

[44] Philip Wadler. Linear types can change the world! In *Programming concepts and methods*, page 561. North-Holland, 1990.

A Definitions

Definition 7. A variable i is unused in P ($\text{UNUSED}_i\ P$) if it appears in none of its inputs and none of its outputs.

$$\frac{i : \text{VAR}_n \qquad P : \text{PROCESS}_n}{\text{UNUSED}_i\ P}\ \textsc{Unused} \qquad \frac{}{\text{UNUSED}_i\ \mathbb{0}}$$

$$\frac{\text{UNUSED}_{1+i}\ P}{\text{UNUSED}_i\ \nu\ P} \qquad \frac{\text{UNUSED}_i\ P \qquad \text{UNUSED}_i\ Q}{\text{UNUSED}_i\ P \parallel Q}$$

$$\frac{i \not\equiv x \qquad \text{UNUSED}_{1+i}\ P}{\text{UNUSED}_i\ x\ (\ )\ P} \qquad \frac{i \not\equiv x \qquad i \not\equiv y \qquad \text{UNUSED}_i\ P}{\text{UNUSED}_i\ x\ \langle y \rangle\ P}$$

The type $i \not\equiv x$ unfolds to the negation of propositional equality on Var, i.e. $i \equiv x \to \bot$.

B Substitution

Theorem 9 (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 to j in P will be well-typed in $\gamma\,;\Gamma_j \vdash P\,[\,i \mapsto j\,] \triangleright \Psi_j$. Refer to Figure 16 for a diagramatic representation.

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 $\Delta_i \equiv \ell_\varnothing$ follows that $m \equiv n$. Therefore $\Gamma_j \equiv \Psi_j$ and end can be applied.
- For constructor ν 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 $\ell_\varnothing$ at index i, there must exist some δ such that $m \coloneqq \delta \cdot^2 n.\,\mathrm{l}$
  - If the input is on the variable being substituted, we split m such that $m \coloneqq \ell_1 \cdot^2 l$ 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 $\Gamma_j \ni_j \ell_1 \triangleright \Xi_j$ 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 $\Gamma \ni_x \ell_1 \triangleright \Theta$ for some Θ. We then construct for some Ξ_j the arrows $\Gamma_j \ni_x \ell_1 \triangleright \Xi_j$ (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.
- $\langle\ \rangle$ applies the ideas outlined for the $(\ )$ constructor to both the VarRef doing the output, and the VarRef for the sent data.
- For $\parallel$ 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.

□