Typing a linear π-calculus

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

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

remove todo notes

Abstract

We present the syntax, operational semantics, and typing rules of a π-calculus with linear and shared types. We use leftover typing [1] to encode our typing rules in a way that propagates linearity constraints into process continuations. We generalize the algebras on multiplicities using indexed sets of *partial commutative monoids*, allowing the user to choose a mix of linear, affine, gradual and shared typing. We provide framing, weakening and strengthening proofs that we then use to prove subject congruence. We show that the type system is stable under substitution and prove subject reduction.

This formalization has been fully mechanized with Agda and is available at `https://github.com/umazalakain/typing-linear-pi`.

2012 ACM Subject Classification Theory of computation → Process calculi

Keywords and phrases pi calculus, linear, types, concurrency

Digital Object Identifier 10.4230/LIPIcs...

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

Acknowledgements I want to thank ...

1 Introduction

The π-calculusmodels communication.

why resource-aware typing

extensional typing rules for a given syntax and operational semantics

leftover typing

1.1 Contribution

Machine verified formalisation of the linear pi calculus

Typing with leftovers applied to the pi calculus

Abstraction over multiplicities

Full formalisation available in Agda

1.2 Notation

$$\frac{}{\mathbb{N} : \mathit{Set}} \qquad \frac{}{0 : \mathbb{N}} \qquad \frac{n : \mathbb{N}}{1{+}n : \mathbb{N}}$$

Figure 1 Notation used in this paper

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Double rule for type-level definitions. We omit universe levels for brevity. Constructors are `teletyped` unless they are symbols. Some constructors are shared across types. They can however always be disambiguated through the type of the goal.

2 Syntax

Abstraction is one of the key reasoning tools in a language: it allows for features to be defined over a range of inputs. Such is the case of the π-calculus too, where both scope restriction and input introduce abstractions. The bodies of these constructs must have a way of referring to their argument. Using names as variable references is a popular option amongst humans. However, names are cumbersome to mechanize: inserting a new variable into an environment means proving that the name of such variable is different to all other variable names in the environment. Moreover, to a machine names are of no significance whatsoever.

Machines prefer things they can algorithmically act upon. Like natural numbers! What does it mean to use a number as a variable reference? The idea de Bruijn had [] was to use the index n to refer to the variable introduced n binders ago. The binders themselves introduce no names anymore. The expression $\lambda g.(\lambda f.fg)g$ in the λ-calculus would translate as $\lambda(\lambda 01)0$. That is, terms at different *depths* must use different indices to refer to the same binding. Humans find this often confusing. There is however no reason not to keep the original names bestowed by the humans together with the indices. Machines can then manipulate references mechanically and still use names to present them to humans.

A variable occurring under n abstractions has n things to refer to. References outside of that range have no associated meaning. It is useful to rule out these nonsensical terms syntactically. In Figure 2 we do so by introducing the indexed family of types VAR_n: for all naturals n, the type VAR_n has n distinct elements.

$$\frac{n : \mathbb{N}}{\mathrm{VAR}_n : \mathit{Set}} \qquad \frac{n : \mathbb{N}}{0 : \mathrm{VAR}_{1+n}} \qquad \frac{x : \mathrm{VAR}_n}{{}_{1+}x : \mathrm{VAR}_{1+n}}$$

Figure 2 Types of size n

Every time we go under a binder, the number of binders a variable might refer to increments by one. To propagate this information, 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. As shown in Figure 3, we increase the *depth* counter every time we create a new channel or receive some input.

Unlike with names, using type-level de Bruijn indices makes our syntax well-scoped by construction. As a consequence, the semantics of our language can be defined on the totality of the syntax. User-friendliness can still be recovered through a function that converts processes with names into (possibly) processes with indices. This function would keep track of what index is associated with what name, and would traverse the process recursively, taking note of new binders and substituting variable references. If the process is ill-scoped, the function would return nothing. To print things back to the user names can be substituted with an index together with the name.

$$\frac{n : \mathbb{N}}{\text{PROCESS}_n : \mathit{Set}}$$

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

Figure 3 Well-scoped grammar using de Bruijn indices

3 Semantics

In the λ-calculus β-reduction operates on syntactically adjacent terms. In the π-calculus wowever, the syntax introduces unnecessary distinctions (e.g. semantically parallel composition is defined modulo associativity and commutativity). There are several ways around this, a structural congruence relation being one of the historical ones. (Others include labeled transition systems and higher inductive types.)

3.1 Structural congruence

Structural congruence is a congruent equivalence relation on processes. Any two structurally congruent processes are strongly bisimilar: they are can follow each other's reduction steps []. Figure 4 lists the base cases of structural congruence.

$$\frac{}{P \equiv Q : \mathit{Set}} \qquad \frac{}{\texttt{comp-assoc} : P \parallel Q \parallel R \equiv P \parallel Q \parallel R} \qquad \frac{}{\texttt{comp-sym} : P \parallel Q \equiv Q \parallel P}$$

$$\frac{}{\texttt{comp-end} : P \parallel \mathbf{0}_n \equiv P} \qquad \frac{}{\texttt{scope-end} : \boldsymbol{\nu}\, \mathbf{0}_{1+n} \equiv \mathbf{0}_n}$$

$$\frac{uQ : \text{UNUSED}_0\, Q}{\texttt{scope-ext} : \boldsymbol{\nu}\, (P \parallel Q) \equiv (\boldsymbol{\nu}\, P) \parallel \mathit{lower}\ 0\ \ Q\ uQ} \qquad \frac{}{\texttt{scope-comm} : \boldsymbol{\nu}\, \boldsymbol{\nu}\, P \equiv \boldsymbol{\nu}\, \boldsymbol{\nu}\, \mathit{swap}\ 0\ P}$$

Figure 4 Structural rewriting rules. Premises P, Q and R are of type PROCESS_n where n can be inferred.

Structural congruence is a congruent equivalence relation ship. As such, rewrites can happen at any point in a process' recursive definition, and they are closed under reflexivity, symmetry and transitivity as shown in Figure 5. In §5.5 we will prove that if two processes P and Q are structurally congruent and P is well-typed, then Q is well-typed. Specifically, in the case of transitivity we must prove that if P is structurally congruent with Q and Q with R, and P is well-typed, then so is R. To do so, we will have to proceed by induction and first get a proof of the well-typedness of Q, then use that to reach R. To show that the doubly recursive call terminates we index the equivalence relation $=$ by the type REC, which models the structure of the recursion.

$$\frac{}{\mathrm{REC} : \mathit{Set}} \qquad \frac{}{\texttt{zero} : \mathrm{REC}} \qquad \frac{r : \mathrm{REC}}{\texttt{one}\ r : \mathrm{REC}} \qquad \frac{r\ s : \mathrm{REC}}{\texttt{two}\ r\ s : \mathrm{REC}}$$

$$\frac{P\ Q : \mathrm{PROCESS}_n \qquad r : \mathrm{REC}}{P =_r Q : \mathit{Set}} \qquad \frac{eq : P \equiv Q}{\texttt{struct}\ eq : P =_{\texttt{zero}} Q}$$

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

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

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

Figure 5 Structural rewriting rules lifted to a congruent equivalence relation indexed by a recursion tree. Premises P, P', Q, and R are of type $\mathrm{PROCESS}_n$ where n can be inferred

congruence relationship indexed by recursive tree

3.2 Reduction

keeping track of the variable on which communication occurs

4 Linear typing rules

4.1 Multiplicities

A type system with both linear and shared resources has multiplicities 0, 1 and ω.

alignment, get rid of $1\cdot$, $\cdot - join$, $\cdot - compute^l$

4.1.1 Example type systems

Shared. Gradual. Affine. Linear.

4.2 Variable references

4.3 Contexts

two-layered approach: types on one hand, capabilities on the other removing from context vs keeping in context but marking it used

4.4 Typing with leftovers

4.4.1 Typing relation

Variable references as proofs of capability

Context splits at each variable reference

$$\frac{n : \mathbb{N}}{\mathrm{CHANNEL}_n : \mathit{Set}} \qquad \frac{}{\texttt{nothing} : \mathrm{CHANNEL}_n} \qquad \frac{i : \mathrm{VAR}_n}{\texttt{just}\ i : \mathrm{CHANNEL}_n}$$

$$\frac{i : \mathrm{CHANNEL}_n \qquad P\ Q : \mathrm{PROCESS}_n}{P \longrightarrow_i Q : \mathit{Set}}$$

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

$$\frac{\mathit{red} : P \longrightarrow_i P'}{\texttt{par}\ \mathit{red} : P \parallel Q \longrightarrow_i P' \parallel Q} \qquad \frac{\mathit{red} : P \longrightarrow_i Q}{\texttt{res}\ \mathit{red} : \boldsymbol{\nu}\, P \longrightarrow_{dec\ i} \boldsymbol{\nu}\, Q}$$

$$\frac{\mathit{eq} : P = P' \qquad \mathit{red} : P' \longrightarrow_i Q}{\texttt{struct}\ \mathit{eq}\ \mathit{red} : P \longrightarrow_i Q}$$

Figure 6 Operational semantics indexed by reducing channel

5 Subject reduction

5.1 Framing

5.2 Weakening

Order preserving embeddings model a series of insertions. We only ever need one insertion to prove subject congruence, but there is no loss of generality.

5.3 Strengthening

5.4 Swapping

5.5 Subject congruence

5.6 Substitution

6 Related work

[?] polymorphic tokens, HOAS

[?]

[?]

[?]

[?]

7 Future work

Work that will be done time permiting:

Affine types

Proof of progress

Product types

$$
\begin{aligned}
\mathtt{0}\cdot &: C \\
+\cdot &: C \\
-\cdot &: C \\
\_ := \_ \cdot \_ &: C \to C \to C \to Set \\
\mathtt{1}\cdot &: C \\
\cdot - \mathtt{join} &: \forall \{xyz\} \to x := y \cdot +\cdot \to x := z \cdot -\cdot \to \exists w.(x := w \cdot 1\cdot) \\
\cdot - \mathtt{compute} &: \forall yz \to Dec(\exists x.(x := y \cdot z)) \\
\cdot - \mathtt{compute}^{\mathtt{l}} &: \forall xz \to Dec(\exists y.(x := y \cdot z)) \\
\cdot - \mathtt{unique} &: \forall \{xx'yz\} \to x' := y \cdot z \to x := y \cdot z \to x' \equiv x \\
\cdot - \mathtt{unique}^{\mathtt{l}} &: \forall \{xyy'z\} \to x := y' \cdot z \to x := y \cdot z \to y' \equiv y \\
\cdot - \mathtt{id}^{\mathtt{l}} &: \forall x \to x := 0 \cdot \cdot x \\
\cdot - \mathtt{comm} &: \forall \{xyz\} \to x := y \cdot z \to x := z \cdot y \\
\cdot - \mathtt{assoc} &: \forall \{xyzuv\} \to x := y \cdot z \to y := u \cdot v \to \exists w.(x := u \cdot w \times w := v \cdot z)
\end{aligned} \tag{1}
$$

Figure 7 Partial commutative monoid

$$
\begin{aligned}
\mathrm{IDX} &: Set \\
\exists\mathrm{IDX} &: Idx \\
\mathrm{CARRIER} &: \mathrm{IDX} \to Set \\
\mathrm{QUANTIFIERS} &: \forall i : \mathrm{IDX} \to \mathrm{QUANTIFIER}_{\mathrm{CARRIER}_i}
\end{aligned} \tag{2}
$$

Figure 8 Indexed set of partial commutative monoids

Sum types

Decidable typechecking

Soundness and completeness with respect to an alternative formalization.

Encoding of session types

References

1 Guillaume Allais. Typing with Leftovers - A mechanization of Intuitionistic Multiplicative-Additive Linear Logic. page 22 pages, 2018. `http://drops.dagstuhl.de/opus/volltexte/2018/10049/`. `doi:10.4230/lipics.types.2017.1`.

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

Figure 9 This is...

$$\frac{n : \mathbb{N}}{\mathrm{IDXS}_n : \mathit{Set}} \qquad \frac{}{[\,] : \mathrm{IDXS}_0} \qquad \frac{is : \mathrm{IDXS}_n \qquad i : \mathrm{IDX}}{is, i : \mathrm{IDXS}_{1+n}}$$

$$\frac{is : \mathrm{IDXS}_n}{\mathrm{CTX}_{is} : \mathit{Set}} \qquad \frac{}{[\,] : \mathit{Ctx}_{[\,]}} \qquad \frac{\Gamma : \mathrm{CTX}_{is} \qquad x : \mathrm{CARRIER}_i}{\Gamma, x : \mathrm{CTX}_{is,i}}$$

Figure 10 This is...

$$\genfrac{}{}{0pt}{}{}{}\frac{\gamma : \mathrm{PRECTX}_n \qquad \begin{array}{c} is : \mathrm{IDXS}_n \\ \Gamma : \mathrm{CTX}_{is} \end{array} \qquad t : \mathrm{TYPE} \qquad x : \mathrm{CARRIER}_i \qquad \Delta : \mathrm{CTX}_{is}}{\gamma \propto \Gamma \ni t \propto x \boxtimes \Delta : \mathit{Set}}$$

$$\frac{\Gamma : \mathrm{CTX}_{is} \qquad y\ z : \mathrm{CARRIER}_i \qquad \mathit{True}(\cdot - \mathtt{compute}\ y\ z)}{\mathtt{zero} : \gamma, t \propto \Gamma, x \ni t \propto y \boxtimes \Gamma, z}$$

$$\frac{\begin{array}{c} \Gamma : \mathrm{CTX}_{is} \qquad x : \mathrm{CARRIER}_i \qquad x' : \mathrm{CARRIER}_j \qquad \Delta : \mathrm{CTX}_{is} \\ \mathit{loc}_x : \gamma \propto \Gamma \ni t \propto x \boxtimes \Delta \end{array}}{\mathtt{suc}\ \mathit{loc}_x : \gamma, t \propto \Gamma, x' \ni t \propto x \boxtimes \Delta, x'}$$

Figure 11 This is...

$$\frac{\gamma : \mathrm{PRECTX}_n \qquad \begin{array}{l} is : \mathrm{IDXS}_n \\ \Gamma : \mathrm{CTX}_{is} \end{array} \qquad P : \mathrm{PROCESS}_n \qquad \Delta : \mathrm{CTX}_{is}}{\gamma \propto \Gamma \vdash P \boxtimes \Delta : Set}$$

$$\frac{}{\texttt{end} : \gamma \propto \Gamma \vdash \mathbf{0} \boxtimes \Gamma} \qquad \frac{\begin{array}{l} t : \mathrm{TYPE} \qquad x : \mathrm{CARRIER}_i \qquad y : \mathrm{CARRIER}_j \\ cont : \gamma, C[t \propto x] \propto \Gamma, y \vdash P \boxtimes \Delta, 0. \end{array}}{\texttt{chan}\ t\ x\ y : \gamma \propto \Gamma \vdash \boldsymbol{\nu}\, P \boxtimes \Delta}$$

$$\frac{\begin{array}{l} chan_x : \gamma \propto \Gamma \ni C[t \propto x] \propto + \cdot \boxtimes \Xi \\ cont : \gamma, t \propto \Xi, x \vdash P \boxtimes \Theta, 0 \cdot \end{array}}{\texttt{recv}\ chan_x\ cont : \gamma \propto \Gamma \vdash toFin\ chan_x\ ()\ P \boxtimes \Theta}$$

$$\frac{\begin{array}{l} chan_x : \gamma \propto \Gamma \ni C[t \propto x] \propto - \cdot \boxtimes \Delta \\ loc_y : \gamma \propto \Delta \ni t \propto x \boxtimes \Xi \\ cont : \gamma \propto \Xi \vdash P \boxtimes \Theta \end{array}}{\texttt{send}\ chan_x\ loc_y\ cont : \gamma \propto \Gamma \vdash toFin\ chan_x\ \langle toFin\ loc_y \rangle\ P \boxtimes \Theta}$$

$$\frac{\begin{array}{l} l : \gamma \propto \Gamma \vdash P \boxtimes \Delta \\ r : \gamma \propto \Delta \vdash Q \boxtimes \Xi \end{array}}{\texttt{comp}\ l\ r : \gamma \propto \Gamma \vdash P \parallel Q \boxtimes \Xi}$$

Figure 12 This is...