Co-Contextual Typing Inference for the Linear π-Calculus in Agda

(Extended Abstract)

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

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

Abstract

A π-calculus with linear types ensures the privacy and safety of concurrent communication. Allowing shared (unlimited) communication however is key to model real-world services. To implement a decision procedure that type checks a π-calculus process with both linear and shared types one can either rely on user-provided type annotations, or infer the types of the channels created within the process. We choose to reduce the burden on the user by following the latter approach. If we limit ourselves to the shared π-calculus, we can traverse a process bottom-up and eagerly solve typing constraints into substitutions and apply them to the typing context. However, in a setting with both linear and shared types, typing constraints do not always come with a most general solution, and thus cannot always be eagerly solved.

We provide a *co-contextual typing inference* [5] algorithm that traverses processes bottom-up and, in addition to the typing context, collects a set of typing constraints. We then solve those constraints that have a most general solution (by using well-known unification algorithms [7]) while deferring the rest until more information becomes available. We state clear soundness and completeness theorems separating both these phases, and a progress theorem that ensures that only those constraints without a most general solution are deferred. This work is being mechanised in Agda.

1 Introduction

The π-calculus [8, 9] models concurrent processing, boiling it down to the transmission of data over communication channels — where channels too are sent as payload. Type systems for the π-calculus that support linearity [6] ensure that linear channels are used *exactly once*, which guarantees communication safety and the absence of race conditions. We follow this line of work with a π-calculus with linear and shared types, where the input and output capabilities of a channel are either usage 0 (cannot be used to transmit), usage 1 (must be used exactly once), or usage ω (unlimited use). [1]

[1]Why support a usage 0 instead of removing the variable from context altogether? it allows the syntax to be independent from the type system, and moreover for a *polarised* treatment of channels, where the same channel variable is used for both input and output.

To type check a π-calculus process with shared and linear types we must assign a type to every communication channel created within the process. To do so we can either ask the user for type annotations [12], or we can infer types by looking at how channels are used. To reduce the burden on the user, we follow the latter approach: we traverse processes bottom-up while keeping a typing context with *metavariables* (*holes*) and collecting typing constraints on metavariables, in line with co-contextual typing [5]. Keeping a strictly bottom-up information flow has the additional advantage of making typing inference parallelisable. Constraints with a most general solution are solved into substitutions and applied to the typing context, constraints without must be kept around for later: applying them as substitutions risks over-constraining the problem down the line. Armed with a typing inference algorithm that for a process P infers the most general typing context Γ and some typing constraints, type checking that $\Delta \vdash P$ for some Δ amounts to emitting the extra constraint $\Gamma = \Delta$, assuming the metavariables in Γ and Δ are disjoint.

Decidable (and certified) type checking procedures have been provided for a range of linear type systems [1–3, 10]. These type systems however do not deal with types containing usage annotations nested within: in our case, a channel with *payload* type "channel with input usage 0" differs in type from a channel with payload type "channel with input usage 1". As a result, we must roll typing inference and linearity inference into one, and deal with the type polymorphism that arises as a result of usage polymorphism. To the best of our knowledge, this problem has not yet been mechanised, and has only been treated in Padovani's work on type reconstruction for composite types [11]. With this work, we aim to:

- State and prove clear soundness and completeness theorems for both constraint collection and constraint resolution (§3).
- Mechanise in Agda the totality of this work.

We start defining an untyped but well scoped π-calculus using type-level de Bruijn indices [4] and embed a small expression language that handles composite sum and product types. On top, we define a standard type system with linear and shared types using context-splits (§2). We then provide an overview of how typing inference works, define constraint collection and constraint resolution (§3), and state that both phases are sound and complete with regards to the type system defined in §2. (Notation: variables are black, types are blue, constructors are green, and functions are gray.)

2 Type System

We define a standard syntax and type system for the linear π-calculus. The only non-standard feature is that types allow for *metavariables* within them.

Syntax We define a standard syntax for the π-calculus using type-level de Bruijn indices. Variable references i_{1+n} are of type $\mathsf{Fin}\ (1+n)$ with constructors zero and $\mathsf{suc}\ i_n$, expressions e_n and f_n are of type $\mathsf{Expr}\ n$, processes p_n and q_n are of type $\mathsf{Proc}\ n$.

$$
\begin{aligned}
e_n\ f_n &:= \mathsf{unit} & p_n\ q_n &:= \mathsf{end} \mid \mathsf{rec}\ p_n \mid \mathsf{new}\ p_{1+n}\\
&\mid \mathsf{var}\ i_n & &\mid \mathsf{recv}\ e_n\ p_{1+n}\\
&\mid \mathsf{pair}\ e_n\ f_n & &\mid \mathsf{send}\ e_n\ f_n\ p_n\\
&\mid \mathsf{fst}\ e_n \mid \mathsf{snd}\ e_n & &\mid \mathsf{comp}\ p_n\ q_n\\
&\mid \mathsf{inl}\ e_n \mid \mathsf{inr}\ e_n & &\mid \mathsf{case}\ e_n\ p_{1+n}\ q_{1+n}
\end{aligned}
$$

Types Both types and usage annotations contain metavariables. To make their handling uniform we use a common set of metavariables for both. A context of kinds γ keeps track of whether a metavariable is of type kind ty or usage annotation kind us. Variable references m are of type $\gamma \ni_{\mathsf{t}} k$: the set of variables of kind k in a kinding context γ. Usage annotations i and o are of type $\gamma \vdash_{\mathsf{t}} \mathsf{us}$, and types s and t of type $\gamma \vdash_{\mathsf{t}} \mathsf{ty}$. We henceforth abbreviate $\gamma \vdash_{\mathsf{t}} \mathsf{ty}$ as $\mathsf{Type}\ \gamma$ and $\gamma \vdash_{\mathsf{t}} \mathsf{us}$ as $\mathsf{Usage}\ \gamma$.

$$i\ o := \mathsf{mvar}_{\mathsf{us}}\ m \mid 0\cdot \mid 1\cdot \mid \omega\cdot$$

$$s\ t := \mathsf{mvar}_{\mathsf{ty}}\ m \mid \mathsf{unit} \mid \mathsf{chan}_{[i,o]}\ t \mid \mathsf{prod}\ s\ t \mid \mathsf{sum}\ s\ t$$

Context Splits A context Γ of type $\mathsf{Ctx}_n\ \gamma$ is a list of $\mathsf{Type}\ \gamma$ of size n. We define context splits $\Gamma = \Delta \uplus \Theta$ pointwise on types. Splits on types are defined pointwise on usage annotations (but are not applied to a channel's payload) and splits on usage annotations are defined as follows:

$$\frac{}{x = x \uplus 0\cdot} \qquad \frac{}{x = 0\cdot \uplus x} \qquad \frac{}{\omega\cdot = x \uplus y}$$

$$\frac{i_x = i_y \uplus i_z \quad o_x = o_y \uplus o_z}{\mathsf{chan}_{[i_x,o_x]}\ t = \mathsf{chan}_{[i_y,o_y]}\ t \uplus \mathsf{chan}_{[i_z,o_z]}\ t}$$

Note that e.g., both $0\cdot = 0\cdot \uplus 0\cdot$ and $\omega\cdot = 0\cdot \uplus 0\cdot$ are possible: we use context splits to allow the type system to lose granularity. Following [11], a context Γ is unrestricted (non-linear) $\mathsf{un}\ \Gamma$ if it can be split into itself: $\Gamma = \Gamma \uplus \Gamma$ (respectively $\mathsf{un}\ x$ and $\mathsf{un}\ t$ for usage annotations x and types t).

Typing Judgments We can now define our typing judgment for variables ($\Gamma \ni i : t$, where variable i under context Γ is of type t), expressions ($\Gamma \vdash e : t$, where expression e under context Γ is well typed with type t) and processes ($\Gamma \vdash p$, where process p is well typed under context Γ). Note that, although fully linear tensor products ($\mathsf{prod}\ s\ t$) cannot directly be eliminated through their projections (fst rule), context splits enable their usage annotations to be distributed beforehand. Similarly, while it may appear that recv and send only work on channels with usages $(1\cdot, 0\cdot)$ and vice versa, the context splits in these rules allow usages to lose granularity (and thus accept e.g., $(\omega\cdot, \omega\cdot)$).

$$\frac{\Gamma : \mathsf{Ctx}_n\ \gamma \quad \mathsf{un}\ \Gamma \quad t : \mathsf{Type}\ \gamma}{\Gamma, t \ni \mathsf{zero} : t} \qquad \frac{\Gamma \ni i : t \quad s : \mathsf{Type}\ \gamma \quad \mathsf{un}\ s}{\Gamma, s \ni \mathsf{suc}\ i : t}$$

$$\frac{\Gamma \vdash e : \mathsf{prod}\ t\ s \quad \mathsf{un}\ s}{\Gamma \vdash \mathsf{fst}\ e : t} \qquad \frac{\Gamma = \Delta \uplus \Theta \quad \Delta \vdash e : s \quad \Theta \vdash f : t}{\Gamma \vdash \mathsf{pair}\ e\ f : \mathsf{prod}\ s\ t}$$

$$\frac{\mathsf{un}\ \Gamma}{\Gamma \vdash \mathsf{end}} \qquad \frac{t : \mathsf{Type}\ \gamma \quad \Gamma, t \vdash p}{\Gamma \vdash \mathsf{new}\ p}$$

$$\frac{\Gamma = \Delta \uplus \Theta \quad \Delta \vdash e : \mathsf{chan}_{[1\cdot,0\cdot]}\ t \quad \Theta, t \vdash p}{\Gamma \vdash \mathsf{recv}\ e\ p}$$

$$\frac{\begin{array}{l}\Gamma = \Delta \uplus \Theta\\ \Theta = \Psi \uplus \Xi \quad \Delta \vdash f : \mathsf{chan}_{[0\cdot,1\cdot]}\ t \quad \Psi \vdash e : t \quad \Xi \vdash p\end{array}}{\Gamma \vdash \mathsf{send}\ f\ e\ p}$$

3 Inference

Constraints Constraints of type $\mathsf{Constr}\ \gamma$ are defined on arguments of type $\gamma \vdash_{\mathsf{t}} k$ for some k — that is, on both usage annotations and types. They take two forms: the binary $[\ S \overset{c}{=} T\ ]$, where S and T must be unified, and the ternary $[\ S \overset{c}{=} T + R\ ]$, where S is split into T and R. Constraints are pointwise lifted to typing contexts, and we abbreviate with $[\ \Gamma \overset{c}{\ni}_i t\ ]$ a constraint that states that i must be of type t in Γ, and all other variables in Γ must be unrestricted. We use $[\mathsf{Constr}\ \gamma\ ]$ to refer to lists of constraints of type $\mathsf{Constr}\ \gamma$.

Substitution A kind-preserving substitution $\mathsf{Subst}\ \gamma\ \delta$ maps usage annotations and type metavariables in γ to usage annotations and types in δ, that is, $\forall\{k\} \to \gamma \ni_{\mathsf{t}} k \to \delta \vdash_{\mathsf{t}} k$ for an implicit k. The function $\sigma \triangleleft t$ of type $\mathsf{Subst}\ \gamma\ \delta \to (\forall\{k\} \to \gamma \vdash_{\mathsf{t}} k \to \delta \vdash_{\mathsf{t}} k)$ performs the substitution by replacing all the metavariables in t with their corresponding terms in σ. Substitutions on constraints are defined pointwise on their arguments.

Constraint Satisfaction We use a $[\![\_]\!]$ function to interpret constraints $[\ S \overset{c}{=} T\ ]$ and $[\ S \overset{c}{=} T + R\ ]$ into claims of their satisfiability $S \equiv T$ and $S = T \uplus R$, respectively.

Inference Typing inference takes a process with n free variables and returns a metavariable context γ, a typing context with n free variables containing metavariables in γ, and a list of constraints on metavariables γ. We define a similar function for typing inference on expressions, this time also returning a type in $\mathsf{Type}\ \gamma$. These functions are total: if a process or expression is untypable its constraints are unsolvable.

$$\mathsf{inferProc} : \mathsf{Proc}\ n \to \exists\gamma.\ \mathsf{Ctx}_n\ \gamma \times [\mathsf{Constr}\ \gamma\ ]$$

$$\mathsf{inferExpr} : \mathsf{Expr}\ n \to \exists\gamma.\ \mathsf{Ctx}_n\ \gamma \times [\mathsf{Constr}\ \gamma\ ] \times \mathsf{Type}\ \gamma$$

Unlike in the shared π-calculus, where constraints have always a most general unifier, in the shared *and* linear π-calculus metavariables can be under-constrained. Consider the open process $\mathsf{send}\ a\ \mathsf{unit}\ (\mathsf{send}\ x\ a\ \mathsf{end})$ where x and a are free: we partly use a to send, then send whatever is left of a away over x and terminate. Let us step through a working example of how inference runs:

1. on end, inference creates a fresh typing context Γ_0 with metavariables for x and a, and constraints demanding that these should be unrestricted.
2. on $\mathsf{send}\ x\ a$, inference creates fresh typing contexts Γ_4, Γ_3, Γ_2 and Γ_1, a fresh metavariable $?t$, and constraints $[\ \Gamma_4 \overset{c}{=} \Gamma_3 + \Gamma_2\ ]$, $[\ \Gamma_3 \overset{c}{\ni}_x \mathsf{chan}_{[0\cdot,1\cdot]}\ ?t\ ]$, $[\ \Gamma_2 \overset{c}{=} \Gamma_1 + \Gamma_0\ ]$, and $[\ \Gamma_1 \overset{c}{\ni}_a\ ?t\ ]$, following the typing rules.
3. on $\mathsf{send}\ a\ \mathsf{unit}$, inference creates fresh typing contexts Γ_6 and Γ_5, and constraints $[\ \Gamma_6 \overset{c}{=} \Gamma_5 + \Gamma_4\ ]$, and $[\ \Gamma_5 \overset{c}{\ni}_a \mathsf{chan}_{[0\cdot,1\cdot]}\ \mathsf{unit}\ ]$.

Here *usage polymorphism* on $?t$ makes inference underconstrained and prevents us from finding a most general solution: the constraints on a's type demand that it must be split into $\mathsf{chan}_{[0\cdot,1\cdot]}\ \mathsf{unit}$, into $?t$, and into some unrestricted leftovers, and while $?t$ can eagerly be substituted by a channel type $\mathsf{chan}_{[?i,?o]}\ \mathsf{unit}$ for some $?i$ and $?o$, it is polymorphic in its usage annotations $?i$ and $?o$. In other words, we must keep track of the partial usage of a while allowing x to be polymorphic in the type of a. As a result, these partial usage constraints must be kept around (potentially until the process is closed and they can be solved by instantiation) and meta-theoretical properties must therefore be abstracted over constraint satisfaction.

Theorem 3.1 (Inference Soundness)**.** Given $\mathsf{infer}\ p$ returns (γ, Γ, cs), every substitution σ satisfying $[\![\sigma \triangleleft cs]\!]$ makes $(\sigma \triangleleft \Gamma) \vdash p$ hold.

Theorem 3.2 (Inference Completeness)**.** Given $\mathsf{infer}\ p$ returns (γ, Γ, cs), for every context Δ such that $\Delta \vdash p$, there exists a substitution σ satisfying $[\![\sigma \triangleleft cs]\!]$ such that Δ is a specialisation of $(\sigma \triangleleft \Gamma)$ — a specialisation $\Delta \subseteq (\sigma \triangleleft \Gamma)$ is defined as $\exists\sigma_f.\ \Delta \equiv (\sigma_f \triangleleft (\sigma \triangleleft \Gamma))$.

3.1 Constraint Resolution

Solving a set of constraints results in a set of substitutions and an unsolved set of simplified constraints where those substitutions have already been applied. The constraints that have been left unsolved do not have a most general solution. Constraints of the form $[\ x \overset{c}{=} y\ ]$ are solved by unification using a kinded version of McBride's unification by structural recursion [7], and have either no solution, or a most general solution that results in a substitution. Constraints of the form $[\ x \overset{c}{=} y + z\ ]$ are solved recursively until a base case is reached, at which point they either have a most general solution or they do not.

$$\mathsf{solve} : [\mathsf{Constr}\ \gamma\ ] \to \mathsf{Subst}\ \gamma\ \delta \times [\mathsf{Constr}\ \delta\ ]$$

Theorem 3.3 (Resolution Soundness)**.** Given $\mathsf{solve}\ cs_1$ returns (σ, cs_2), every substitution σ_f that satisfies the simplified constraints ($[\![\sigma_f \triangleleft cs_2]\!]$) satisfies the original constraints after substitutions are applied ($[\![\sigma_f \triangleleft (\sigma \triangleleft cs_1)]\!]$).

Theorem 3.4 (Resolution Completeness)**.** Given $\mathsf{solve}\ cs_1$ returns (σ, cs_2), any substitution σ_f that makes the original constraints cs_1 hold ($[\![\sigma_f \triangleleft cs_1]\!]$) can be decomposed into σ followed by a certain σ_g ($\sigma_f \doteq \sigma_g \cdot \sigma$) that makes the returned constraints cs_2 hold ($[\![\sigma_g \triangleleft cs_2]\!]$).

Theorem 3.5 (Resolution Progress)**.** Given $\mathsf{solve}\ cs_1$ returns (σ, cs_2), to keep us from returning the original constraints as output (which is both sound and complete), we postulate that none of the constraints $c \in cs_2$ have a most general solution, where a most general solution for a constraint c is defined as $\exists\sigma.\ [\![\sigma \triangleleft c]\!] \times (\forall\sigma_f.\ [\![\sigma_f \triangleleft c]\!] \times (\exists\sigma_g.\ \sigma_f \doteq \sigma_g \cdot \sigma))$.

4 Conclusion

We have outlined a procedure for decidable type checking and inference of a π-calculus with linear and shared types. Constraint collection and constraint resolution are kept separate, and their metatheory allows for deferred constraints. We have proved in Agda the soundness of constraint collection 3.1 and of equality constraint resolution 3.3, the remaining proofs are still in progress.

References

[1] G. Allais. Typing with leftovers - A mechanization of intuitionistic multiplicative-additive linear logic. In A. Abel, F. N. Forsberg, and A. Kaposi, editors, *23rd International Conference on Types for Proofs and Programs, TYPES 2017, May 29-June 1, 2017, Budapest, Hungary*, volume 104 of *LIPIcs*, pages 1:1–1:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi: 10.4230/LIPIcs.TYPES.2017.1. URL https://doi.org/10.4230/LIPIcs.TYPES.2017.1.

[2] G. Allais and C. McBride. Certified proof search for intuitionistic linear logic. 2015.

[3] E. C. Brady. Idris 2: Quantitative type theory in practice. In A. Møller and M. Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 9:1–9:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi: 10.4230/LIPIcs.ECOOP.2021.9. URL https://doi.org/10.4230/LIPIcs.ECOOP.2021.9.

[4] N. Bruijn, de. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75 (5):381–392, 1972. ISSN 1385-7258. doi: 10.1016/1385-7258(72)90034-0.

[5] S. Erdweg, O. Bracevac, E. Kuci, M. Krebs, and M. Mezini. A co-contextual formulation of type rules and its application to incremental type checking. In J. Aldrich and P. Eugster, editors, *Proceedings of*

the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015, pages 880–897. ACM, 2015. doi: 10.1145/2814270.2814277. URL https://doi.org/10.1145/2814270.2814277.

[6] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. In H. Boehm and G. L. S. Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 358–371. ACM Press, 1996. doi: 10.1145/237721.237804. URL https://doi.org/10.1145/237721.237804.

[7] C. McBride. First-order unification by structural recursion. *J. Funct. Program.*, 13(6):1061–1075, 2003. doi: 10.1017/S0956796803004957. URL https://doi.org/10.1017/S0956796803004957.

[8] R. Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999. ISBN 978-0-521-65869-0.

[9] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992. doi: 10.1016/0890-5401(92)90008-4. URL https://doi.org/10.1016/0890-5401(92)90008-4.

[10] D. Orchard, V. Liepelt, and H. E. III. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.*, 3(ICFP): 110:1–110:30, 2019. doi: 10.1145/3341714. URL https://doi.org/10.1145/3341714.

[11] L. Padovani. Type reconstruction for the linear π-calculus with composite regular types. *Log. Methods Comput. Sci.*, 11(4), 2015. doi: 10.2168/LMCS-11(4:13)2015. URL https://doi.org/10.2168/LMCS-11(4:13)2015.

[12] U. Zalakain and O. Dardha. π with leftovers: A mechanisation in agda. volume 12719 of *Lecture Notes in Computer Science*, pages 157–174. Springer, 2021. doi: 10.1007/978-3-030-78089-0_9. URL https://doi.org/10.1007/978-3-030-78089-0_9.