Co-Contextual Typing Inference for the Linear \( \pi \)-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 \( \pi \)-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 \( \pi \)-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 \( \pi \)-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 [2] 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 [4]) 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 \( \pi \)-calculus [5, 6] models concurrent processing, boiling it down to the transmission of data over communication channels — where channels too are sent as payload. The linear \( \pi \)-calculus [3] introduces a type system that ensures that every channel is used exactly once. This restriction ensures communication privacy, communication safety, and the absence of race conditions. We generalise to a \( \pi \)-calculus with linear and shared types, where the input and output multiplicities of a channel are either 0 (cannot be used to transmit), 1 (must be used exactly once), or \( \omega \) (unlimited use).

To type check a \( \pi \)-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 [8], 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 in line with co-contextual typing [2]. Keeping a strictly bottom-up information flow has the additional advantage of making typing inference easily parallelisable. Constraints with a most general solution are solved into substitutions and applied to the typing context, constraints without one 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 \( \Gamma \) and some typing constraints, type checking that \( \Delta \vdash P \) for some \( \Delta \) amounts to emitting the extra constraint \( \Gamma = \Delta \), assuming the metavariables in \( \Gamma \) and \( \Delta \) are disjoint.

To the best of our knowledge, this problem has only been treated in Padovani’s work on type reconstruction for composite types [7]. However, his work does not appear to be mechanised, and metatheoretical properties like soundness and completeness are only informally addressed. With this work, we aim to:

- State and prove clear soundness and completeness theorems for both constraint collection (§3.1) and constraint resolution (§3.2).
- Mechanise in Agda the totality of this work.
We start defining an untyped but well scoped π-calculus using type-level de Bruijn indices [1] 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 (§3.1) and constraint resolution (§3.2), and state that both phases are sound and complete with regards to the type system defined in §2. (A note on notation: variables are black, types are blue, constructors are green, functions are gray, and we are sorry this is not a poem.)

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 \( i_n \) are of type \( \text{Fin} n \), expressions \( e_n \) and \( f_n \) are of type \( \text{Expr} n \), processes \( p_n \) and \( q_n \) are of type \( \text{Proc} n \).

\[
e_n f_n := \text{unit} \quad p_n q_n := \text{end} | \text{rec} p_n | \text{new} p_{n+1} \\
| \text{var} i_n \quad | \text{recv} e_n p_{n+1} \\
| \text{pair} e_n f_n \quad | \text{send} e_n f n \\
| \text{fst} e_n | \text{snd} e_n \quad | \text{comp} p_n q_n \\
| \text{inl} e_n | \text{inr} e_n \quad | \text{case} e_n p_{n+1} q_{n+1}
\]

Types. Both types and usage annotations contain metavariables. We use a common set of metavariables for both — this makes their handling uniform. A context of kinds \( \gamma \) keeps track of whether a metavariable is of type kind \( \text{ty} \) or usage annotation kind \( \text{us} \). We refer to a metavariable of kind \( k \) in a kinding context \( \gamma \) as \( \gamma \vdash k \). We can now define usage annotations and types in one go (we define \( \text{th}, \text{l}, \text{w} \) at once for brevity):

\[
\begin{align*}
m &: \gamma \vdash k & i &: \gamma \vdash \text{ty} | us & o &: \gamma \vdash \text{ty} & us & t &: \gamma \vdash \text{ty} \\
\text{mvar} m &: \gamma \vdash k & \text{chan} i o t &: \gamma \vdash \text{ty} \\
\text{unit} &: \gamma \vdash \text{ty} & \text{prod} s &: \gamma \vdash \text{ty} \\
\sum s &: \gamma \vdash \text{ty} & t &: \gamma \vdash \text{ty} \\
\text{sum \ s \ t &: \gamma \vdash \text{ty} \ \ t &: \gamma \vdash \text{ty} & \text{us}
\end{align*}
\]

We henceforth use \( \text{Type} \; \gamma \) for \( \gamma \vdash \text{ty} \) and \( \text{Usage} \; \gamma \) for \( \gamma \vdash \text{us} \).

Context Splits. A context \( \Gamma \) of type \( \text{Ctx} n \; \gamma \) is a list of \( \text{Type} \; \gamma \) of size \( n \). We define context splits \( \Gamma = \Delta \uplus \Theta \) pointwise on types. Splits on types are defined recursively on usage annotations — note that only the usage annotations at the top of a channel are split, not those within its payload. Splits on usage annotations are defined as follows:

\[
x = x \uplus \emptyset \cdot \quad x = \emptyset \cdot x \quad \omega = x \uplus y
\]

Following [7], a context \( \Gamma \) is unrestricted (non-linear) \( \uplus \) if it can be split into itself: \( \Gamma = \Gamma \uplus \Gamma \) (respectively \( \emptyset \cdot x \) and \( \uplus t \) for usage annotations \( x \) and types \( t \)).

Typing Judgments. We can finally define our typing judgment for variables, expressions and processes (some cases are omitted for brevity).

\[
\begin{align*}
\Gamma &: \text{Ctx} n \; \gamma \quad \text{un} \quad \Gamma \vdash t &: \text{Type} \; \gamma \\
\gamma \\
\eta \\
\Gamma, t & \ni z e \ni \text{prod} \quad s \ni \text{un} \\
\Gamma & \ni \text{var} i \ni t \\
\Gamma & \ni \text{fst} e \ni t \\
\Gamma & \ni \text{inl} e \ni \text{sum} \ s \\
\Gamma, \emptyset & \ni \text{end} \\
\Gamma & \ni \text{rec} p \\
\Gamma & \ni \text{new} p \\
\Gamma & \ni \text{case} e \ni p \quad q
\end{align*}
\]

3 Inference

Co-contextual typing inference traverses processes bottom-up keeping a typing context and collecting constraints. This makes inference subproblems independent and thus easily parallelisable. In the shared π-calculus, inference constraints always have a most general unifier, in the shared and linear π-calculus however they do not. Consider the open process \( \text{send} a \ni \text{unit} (\text{send} x \ni \text{end}) \) where \( x \) and \( a \) are free: we partly use \( a \) to send, then send \( a \) away over \( x \) and terminate. When the process terminates, all multiplicities must be unrestricted, but we cannot decide whether to instantiate with \( \emptyset \cdot \) or \( \omega \cdot \). Similarly, we ignore how much of \( a \) we are sending away over \( x \): that depends on what the receiving process expects. Nonetheless, we do have to keep track of the fact that \( a \) has been used once to send.

In §3.1 we introduce typing constraints, and define kind-preserving substitution, constraint satisfaction, and a typing inference algorithm. We show that satisfying the constraints generated by typing inference is enough to make a process typeable (soundness), and postulate that for any typeable process, typing inference will find the most general set of constraints that makes the process typeable (completeness).

In §3.2 we solve equality constraints by unification and context split constraints by decomposition. Constraints with a most general solution can be solved by instantiation once the process is closed. Solving a set of constraints thus results in a set of substitutions and a
of simplified constraints that had those substitutions applied. We postulate that satisfying the simplified constraints amounts to satisfying the original constraints after substitution (soundness), and that for every substitution that solves the original constraints we can find a most general substitution that will solve the simplified constraints (completeness). Additionally, we postulate that the simplified set of constraints only contains constraints without a most general solution (progress).

3.1 Constraint Collection

Constraints. Constraints of type \(\text{Constr } \gamma\) are defined on arguments of type \(\gamma \vdash t \ k\) for some \(k\) — that is, on both usage annotations and types. They take two forms: the binary [\(S \cong T\)], where \(S\) and \(T\) must be unified, and the ternary [\(S \cong T + R\)], where \(T\) and \(R\) must add up to \(S\). Constraints of the former form can be eagerly solved, while constraints of the latter can sometimes not. We use \(\lfloor \text{Constr } \gamma \rfloor\) to refer to lists of constraints of type \(\text{Constr } \gamma\).

Substitution. A kind-preserving substitution \(\text{Subst } \gamma \delta\) maps usage annotations and type metavariables in \(\gamma\) to usage annotations and types in \(\delta\), that is, \(\forall k \rightarrow \gamma \exists_t k \rightarrow \delta\). The function \(\sigma \vdash t\) of type \(\text{Subst } \gamma \delta \rightarrow (\forall k \rightarrow \gamma \vdash t \ k \rightarrow \delta\)} performs the substitution by replacing all the metavariables in \(t\) with their corresponding terms in \(\sigma\). Substitutions on constraints are defined pointwise on their arguments.

Constraint Satisfaction. We use a \(\lfloor \ldots \rfloor\) function to interpret constraints \([S \cong T]\) and \([S \cong T + R]\) into their reified counterparts \(S \equiv T\) and \(S \equiv T \cup R\), respectively.

Inference. Typing inference takes a process with \(n\) free variables and returns a metavariable context \(\gamma\), a typing context with \(n\) free variables containing metavariables in \(\gamma\), and a list of constraints on metavariables \(\gamma\). We define a similar function for typing inference on expressions, this time also returning a type \(\tau\) of type \(\gamma\):

- \(\text{inferProc} : \text{Proc } n \rightarrow \exists \ y. \text{Ctx}_n \ y \times [\text{Constr } \gamma]\)
- \(\text{inferExpr} : \text{Expr } n \rightarrow \exists \ y. \text{Ctx}_n \ y \times [\text{Constr } \gamma] \times \text{Type } \gamma\)

These functions are total: if a process or expression is typable its constraints will be unsolvable.

Let us provide a couple of examples illustrating how this function is defined: For the expression \(f s t \ e\) we 1) recursively infer \(e\) resulting in \(\Gamma, \Gamma, \text{C}(s, t, \tau)\); 2) add two fresh type metavariables \(?t\) and \(?s\) to \(\gamma\); 3) add constraints that \(t \equiv \text{prod } ?t \ ?s\) and \(\ ?s \equiv ?s + ?s\) to \(\text{C}(s)\); and 4) return \(?t\) as the type of \(f s t\).

For the expression \(c a s e \ e \ p \ q\) we 1) infer \(p\) resulting in \(\theta_p, (\Theta_p, s), \text{C}(p, 2)\); 2) infer \(q\) resulting in \(\theta_q, (\Theta_q, t), \text{C}(q, 3)\); 3) infer \(e\) resulting in \(\Delta_e, \text{C}(e, \tau)\); 4) create a new metavariable \(\gamma\) and a new typing context \(\Gamma\) where every type is a fresh metavariable in \(\gamma\); 5) take the union of the metavariables in \(\gamma, \theta_p, \theta_q\) and \(\delta_e\); 6) take the union of the constraints in \(\text{C}(p, s), \text{C}(q, t)\), and \(\text{C}(e, \tau)\), and create new constraints \(\Theta_p \equiv \Theta_q\), \(\Gamma \equiv \Delta_e + \Theta_p\), and \(\tau \equiv \text{sum } s \ t\); and 7) return \(\Gamma\) as the inferred context.

Theorem 3.1 (Inference Soundness). Given \(\text{infer } \gamma, \Gamma, cs\) every substitution \(\sigma\) that makes the constraints \(cs\) hold makes \((\sigma \vdash \Gamma) \vdash p\) hold.

Theorem 3.2 (Inference Completeness). Given \(\text{infer } \gamma, \Gamma, cs\), for every context \(\Delta\) that makes \(p\) typable, there exists a substitution \(\sigma\) that makes the constraints \(cs\) such that \(\Delta\) is a specialisation of \((\sigma \vdash \Gamma)\) — a specialisation \(\Delta \subseteq (\sigma \vdash \Gamma)\) is defined as \(\exists \sigma f. \Delta \equiv (\sigma f \cdot (\sigma \vdash \Gamma))\).

3.2 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 \([x \equiv \gamma + z]\) are solved by unification using a kinded version of McBride’s unification by structural recursion [4], and have either no solution, or a most general solution that results in a substitution. Constraints of the form \([x \equiv \gamma + z]\) are solved recursively until a base case is reached, at which point they either have a most general solution or they do not.

\[\text{solve} : \lfloor \text{Constr } \gamma \rfloor \rightarrow \text{Subst } \gamma \delta \times \lfloor \text{Constr } \delta \rfloor\]

Theorem 3.3 (Resolution Soundness). Given \(\text{solve } cs_1\) returns \((\sigma, cs_2)\), every substitution \(\sigma f\) that satisfies the simplified constraints \(\lfloor \sigma f \equiv cs_2\rfloor\) satisfies the original constraints after substitutions are applied \(\lfloor \sigma f \equiv (\sigma \vdash cs_1)\rfloor\).

Theorem 3.4 (Resolution Completeness). Given \(\text{solve } cs_1\) returns \((\sigma, cs_2)\), any substitution \(\sigma f\) that makes the original constraints \(cs_1\) hold \(\lfloor \sigma f \equiv cs_2\rfloor\) can be decomposed into \(\sigma\) followed by a certain \(\sigma g\) \((\sigma f \equiv \sigma g \cdot \sigma)\) that makes the returned constraints \(cs_2\) hold \(\lfloor \sigma g \equiv cs_2\rfloor\).

Theorem 3.5 (Resolution Progress). Given \(\text{solve } cs_1\) returns \((\sigma, cs_2)\), to keep us from returning the original constraints as output (which is both sound and complete), we postulate that none of the constraints \(c \in cs_2\) has a most general solution, where a most general solution for a constraint \(c\) is defined as \(\exists \sigma. \lfloor c \equiv \sigma \rfloor \times \forall \sigma f. \lfloor \sigma f \equiv c \rfloor \times (\exists \sigma g. \sigma f \equiv \sigma g \cdot \sigma)\).

4 Discussion

We have outlined a procedure for decidable type checking and inference of a \(\pi\)-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


