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 [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 \(\pi\)-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 \(\pi\)-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 \(\pi\)-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 \(\omega\) (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 \(\pi\)-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 [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 \(\Gamma\) 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.

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 \(\pi\)-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
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 \( \text{Fin}(1+n) \) with constructors \( \text{zero} \) and \( \text{suc} i_n \), expressions \( e_n \) and \( f_n \) are of type \( \text{Expr} n \), processes \( p_n \) and \( q_n \) are of type \( \text{Proc} n \).

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

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 \( \gamma \) keeps track of whether a metAVARiable is of type kind \( t \) or usage annotation kind \( \text{us} \). Variable references \( m \) are of type \( \gamma \ni t \): the set of variables of kind \( k \) in a kinding context \( \gamma \). Usage annotations \( i \) and \( o \) are of type \( \gamma \ni t \), and types \( s \) and \( t \) of type \( \gamma \ni t \). We henceforth abbreviate \( \gamma \ni t \) as \( Type \gamma \) and \( \gamma \ni t \) as \( Usage \gamma \).

\[
i_0 := \text{mvar} \text{us} m \mid \theta \mid \text{one} \mid \omega \\
s t := \text{mvar} \text{ty} m \mid \text{unit} \mid \text{chan}[i,o] t \mid \text{prod} s t \mid \text{sum} s t
\]

Context Splits A context \( \Gamma \) of type \( \text{Ctx} n \gamma \) is a list of \( Type \gamma \) 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:

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

\[
i_x = i_y \uplus i_z \quad o_x = o_y \uplus o_z \quad t = \text{chan}[i_x,o_x] t \quad \text{chan}[i_x,o_x] t
\]

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

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

\[
\begin{align*}
  \Gamma : \text{Ctx} n \gamma & \quad un \Gamma \mid t : \text{Type} \gamma \\
  \Gamma, t \ni \text{zero} : t & \quad \Gamma, s \ni \text{suc} i : s \\
  \Gamma \ni e : \text{prod} s t \quad \text{un} s & \quad \Gamma = \Delta \uplus \Theta \quad \Gamma \ni e : s \Theta \ni f : t \\
  \Gamma \ni \text{fst} e : t & \quad \Gamma \ni \text{fist} e : t \\
  \Gamma \ni \text{end} & \quad \Gamma \ni \text{new} p \\
  \Gamma \ni \text{pair} e : \text{prod} s t & \quad \Gamma \ni \text{recv} e p
\end{align*}
\]

3 Inference

Constraints Constraints of type \( \text{Constr} \gamma \) are defined on arguments of type \( \gamma \ni t \) for some \( k \) — that is, on both usage annotations and types. They take two forms: the binary [\( S \leq T \)], where \( S \) and \( T \) must be unified, and the ternary [\( S \leq T \uplus R \)], where \( S \) is split into \( T \) and \( R \). Constraints are pointwise lifted to typing contexts, and we abbreviate with \( \Gamma \ni \Delta \ni t \) a constraint that states that \( \Delta \) must be of type \( t \) in \( \Gamma \), and all other variables in \( \Gamma \) must be unrestricted. We use \( \text{Constr} \gamma \) 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, \( \forall k \rightarrow \gamma \uplus \delta \rightarrow k \uplus \delta \) for an implicit \( k \). The function \( \sigma \ni t \) of type \( \text{Subst} \gamma \delta \rightarrow (\forall k \rightarrow \gamma \uplus \delta \rightarrow k \uplus \delta \) 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 \( \langle \_ , \_ \rangle \) function to interpret constraints [\( S \leq T \)] and [\( S \leq T \uplus R \)] into claims of their satisfiability [\( S \equiv T \uplus T \uplus R \)], 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 in \( \text{Type } \gamma \). These functions are
total: if a process or expression is untypable its constraints
are unsolvable.

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

Unlike in the shared \( \pi \)-calculus, where constraints have
always a most general unifier, in the shared and linear \( \pi \)-
calculus metavariables can be under-constrained. Consider
the open process \( \text{send a unit } (\text{send } x \ a \ \text{end}) \) 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 \( \text{end} \), inference creates a fresh typing context \( \Gamma_0 \) with
metavariables for \( x \) and \( a \), and constraints demanding
that these should be unrestricted.
2. on \( \text{send } x \ a \), inference creates fresh typing contexts \( \Gamma_1 \),
\( \Gamma_2 \), and \( \Gamma_3 \), a fresh metavariable \( ?t \), and constraints
\( \Gamma_1 \triangleleft \Gamma_2 + \Gamma_3 \), \( \Gamma_3 \triangleleft \chi \text{chan}[0,1,1] \ ?t \), \( \Gamma_2 \triangleleft \Gamma_1 + \Gamma_3 \),
and \( \Gamma_1 \triangleleft \Gamma_3 \ ?t \), following the typing rules.
3. on \( \text{send } a \ \text{unit} \), inference creates fresh typing contexts \( \Gamma_5 \) and \( \Gamma_6 \), and constraints \( \Gamma_6 \triangleleft \Gamma_5 + \Gamma_4 \), and
\( \Gamma_1 \triangleleft \chi \text{chan}[0,1,1] \ \text{unit} \).

Here usage polymorphism on \( ?t \) makes inference under-
constrained and prevents us from finding a most general solution: the
constraints on \( a \)’s type demand that it must be split into
\( \chi \text{chan}[0,1,1] \ \text{unit} \), into \( ?t \), and into some unre-
stricted leftovers, and while \( ?t \) can eagerly be substituted by a
channel type \( \chi \text{chan}[7,0,0] \ \text{unit} \) for some \( ?i \) and \( ?o \), it is poly-
morphic 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 \( \text{infer } p \) returns
\( (\gamma, \Gamma, \text{cs}) \), every substitution \( \sigma \) satisfying \( [\sigma \triangleleft \text{cs}] \) makes
(\( \sigma \triangleleft \Gamma \)) \( \vdash p \) hold.

**Theorem 3.2 (Inference Completeness).** Given \( \text{infer } p \) returns
\( (\gamma, \Gamma, \text{cs}) \), for every context \( \Delta \) such that \( \Delta \vdash p \), there
exists a substitution \( \sigma \) satisfying \( [\sigma \triangleleft \text{cs}] \) such that \( \Delta \) is a
specialisation of \( (\sigma \triangleleft \Gamma) \) — a specialisation \( \Delta \triangleleft (\sigma \triangleleft \Gamma) \) is
defined as \( \exists \Delta \Delta \equiv (\sigma \triangleleft (\sigma \triangleleft \Gamma)) \).

### 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 \( [ x \triangleleft y ] \) 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
\( [ x \triangleleft y + z ] \) are solved recursively until a base case
is reached, at which point they either have a most general solution or they do not.

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

**Theorem 3.3 (Resolution Soundness).** Given \( \text{solve } cs_1 \) returns
\( (\sigma, cs_2) \), every substitution \( \sigma_f \) 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 \( \text{solve } cs_1 \) returns
\( (\sigma, cs_2) \), any substitution \( \sigma_f \) that makes the original
constraints \( cs_1 \) hold \( (\| \sigma_f \triangleleft cs_1 \|) \) can be decomposed into \( \sigma \)
solved by a certain \( \sigma_g \text{ s.t. } (\sigma_f \triangleleft \sigma_g \cdot \sigma) \) that makes the returned
constraints \( cs_2 \) hold \( (\| \sigma_g \triangleleft cs_2 \|) \).

**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 \( c \in cs_2 \) have a most gen-
eral solution, where a most general solution for a constraint \( c \)
\( \equiv \exists \Delta \Delta \equiv (\sigma \triangleleft c) \times (\| \sigma_f \triangleleft c \|) \times (\exists \sigma_g \Delta \sigma_g \equiv \sigma_f \cdot \sigma) \).

### 4 Conclusion

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 sepa-
rate, and their metatheory allows for deferred constraints.
We have proved in Agda the soundness of constraint collec-
tion 3.1 and of equality constraint resolution 3.3, the remain-
ing proofs are still in progress.

### References

multiplicative-additive linear logic. In A. Abel, F. N. Forsberg, and
A. Kapoali, 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 -

linear logic. 2015.

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 -

[4] N. Bruin, de. Lambda calculus notation with nameless dummies,
a tool for automatic formula manipulation, with application to the

[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. Eusteg, editors, Proceedings of


