

TYPE RECONSTRUCTION FOR THE LINEAR π -CALCULUS WITH COMPOSITE REGULAR TYPES*

LUCA PADOVANI

Dipartimento di Informatica, Università di Torino, Italy
e-mail address: luca.padovani@di.unito.it

ABSTRACT. We extend the linear π -calculus with composite regular types in such a way that data containing linear values can be shared among several processes, if there is no overlapping access to such values. We describe a type reconstruction algorithm for the extended type system and discuss some practical aspects of its implementation.

1. INTRODUCTION

The linear π -calculus [15] is a formal model of communicating processes that distinguishes between *unlimited* and *linear* channels. Unlimited channels can be used without restrictions, whereas linear channels can be used for one communication only. Despite this seemingly severe restriction, there is evidence that a significant portion of communications in actual systems take place on linear channels [15]. It has also been shown that structured communications can be encoded using linear channels and a continuation-passing style [13, 3]. The interest in linear channels has solid motivations: linear channels are efficient to implement, they enable important optimizations [9, 8, 15], and communications on linear channels enjoy important properties such as interference freedom and partial confluence [18, 15]. It follows that understanding whether a channel is used linearly or not has a primary impact in the analysis of systems of communicating processes.

Type reconstruction is the problem of *inferring* the type of entities used in an unannotated (*i.e.*, untyped) program. In the case of the linear π -calculus, the problem translates into understanding whether a channel is linear or unlimited, and determining the type of messages sent over the channel. This problem has been addressed and solved in [10]. The goal of our work is the definition of a type reconstruction algorithm for the linear

2012 ACM CCS: [Theory of computation]: Models of computation; Semantics and reasoning—Program constructs; [Software and its engineering]: Software notations and tools—General programming languages—Language features.

Key words and phrases: linear pi-calculus, composite regular types, shared access to data structures with linear values, type reconstruction.

* A preliminary version of this paper [20] appears in the proceedings of the 17th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'14).

This work has been supported by ICT COST Action IC1201 BETTY, MIUR project CINA, Ateneo/CSP project SALT, and the bilateral project RS13MO12 DART.

π -calculus extended with pairs, disjoint sums, and possibly infinite types. These features, albeit standard, gain relevance and combine in non-trivial ways with the features of the linear π -calculus. We explain why this is the case in the rest of this section.

The term below

$$\text{*succ?}(x, y).y!(x + 1) \mid \text{new } a \text{ in } (\text{succ!}(39, a) \mid a?(z).\text{print!}z) \quad (1.1)$$

models a program made of a persistent service (the `*`-prefixed process waiting for messages on channel `succ`) that computes the successor of a number and a client (the `new`-scoped process) that invokes the service and prints the result of the invocation. Each message sent to the service is a pair made of the number x and a continuation channel y on which the service sends the result of the computation back to the client. There are three channels in this program, `succ` for invoking the service, `print` for printing numbers, and a private channel a which is used by the client for receiving the result of the invocation. In the linear π -calculus, types keep track of how each occurrence of a channel is being used. For example, the above program is well typed in the environment

$$\text{print} : [\text{int}]^{0,1}, \text{succ} : [\text{int} \times [\text{int}]^{0,1}]^{\omega,1}$$

where the type of `print` indicates not only the type of messages sent over the channel (`int` in this case), but also that `print` is never used for input operations (the 0 annotation) and is used once for one output operation (the 1 annotation).

The type of `succ` indicates that messages sent over `succ` are pairs of type $\text{int} \times [\text{int}]^{0,1}$ – the service performs exactly one output operation on the channel y which is the second component of the pair – and that `succ` is used for an unspecified number of input operations (the ω annotation) and exactly one output operation (the 1 annotation). Interestingly, the overall type of `succ` can be expressed as the combination of two slightly different types describing how each occurrence of `succ` is being used by the program: the leftmost occurrence of `succ` is used according to the type $[\text{int} \times [\text{int}]^{0,1}]^{\omega,0}$ (arbitrary inputs, no outputs), while the rightmost occurrence of `succ` is used according to the type $[\text{int} \times [\text{int}]^{0,1}]^{0,1}$ (no inputs, one output). Following [15], we capture the overall use of a channel by means of a *combination operator* $+$ on types such that, for example,

$$[\text{int} \times [\text{int}]^{0,1}]^{\omega,0} + [\text{int} \times [\text{int}]^{0,1}]^{0,1} = [\text{int} \times [\text{int}]^{0,1}]^{\omega,1}$$

Concerning the restricted channel a , its rightmost occurrence is used according to the type $[\text{int}]^{1,0}$, since there a is used for one input of an integer number; the occurrence of a in $(39, a)$ is in a message sent on `succ`, and we have already argued that the service uses this channel according to the type $[\text{int}]^{0,1}$; the type of the leftmost, binding occurrence of a is the combination of these two types, namely:

$$[\text{int}]^{0,1} + [\text{int}]^{1,0} = [\text{int}]^{1,1}$$

The type of a indicates that the program performs exactly one input and exactly one output on a , hence a is a linear channel. Since a is restricted in the program, even if the program is extended with more processes, it is not possible to perform operations on a other than the ones we have tracked in its type.

The key ingredient in the discussion above is the notion of type combination [15, 10, 24], which allows us to gather the overall number of input/output operations performed on a channel. We now discuss how type combination extends to composite and possibly infinite types, which is the main novelty of the present work.

So far we have taken for granted the ability to perform *pattern matching* on the message received by the service on `succ` and to assign distinct names, x and y , to the components of the pair being analyzed. Pattern matching is usually compiled using more basic operations. For example, in the case of pairs these operations are the `fst` and `snd` projections that respectively extract the first and the second component of the pair. So, a low-level modeling of the successor service that uses `fst` and `snd` could look like this:

$$*\text{succ?}(p) . \text{snd}(p) ! (\text{fst}(p) + 1) \quad (1.2)$$

This version of the service is operationally equivalent to the previous one, but from the viewpoint of typing there is an interesting difference: in (1.1) the two components of the pair are given distinct names x and y and each name is used *once* in the body of the service; in (1.2) there is only one name p for the whole pair which is projected *twice* in the body of the service. Given that each projection accesses only one of the two components of the pair and ignores the other, we can argue that the occurrence of p in `snd`(p) is used according to the type $\text{int} \times [\text{int}]^{0,1}$ (the 1 annotation reflects the fact that the second component of p is a channel used for an output operation) whereas the occurrence of p in `fst`(p) is used according to the type $\text{int} \times [\text{int}]^{0,0}$ (the second component of p is not used). The key idea, then, is that we can extend the type combination operator $+$ component-wise to product types to express the overall type of p as the combination of these two types:

$$(\text{int} \times [\text{int}]^{0,1}) + (\text{int} \times [\text{int}]^{0,0}) = (\text{int} + \text{int}) \times ([\text{int}]^{0,1} + [\text{int}]^{0,0}) = \text{int} \times [\text{int}]^{0,1}$$

According to the result of such combination, the second component of p is effectively used only once despite the multiple syntactic occurrences of p .

The extension of type combination to products carries over to disjoint sums and also to infinite types as well. To illustrate, consider the type t_{list} satisfying the equality

$$t_{\text{list}} = \text{Nil} \oplus \text{Cons}([\text{int}]^{1,0} \times t_{\text{list}})$$

which is the disjoint sum between `Nil`, the type of empty lists, and `Cons`($[\text{int}]^{1,0} \times t_{\text{list}}$), the type of non-empty lists with head of type $[\text{int}]^{1,0}$ and tail of type t_{list} (we will see shortly that there is a unique type t_{list} satisfying the above equality relation). Now, t_{list} can be expressed as the combination $t_{\text{odd}} + t_{\text{even}}$, where t_{odd} and t_{even} are the types that satisfy the equalities

$$t_{\text{odd}} = \text{Nil} \oplus \text{Cons}([\text{int}]^{1,0} \times t_{\text{even}}) \quad \text{and} \quad t_{\text{even}} = \text{Nil} \oplus \text{Cons}([\text{int}]^{0,0} \times t_{\text{odd}}) \quad (1.3)$$

(again, there are unique t_{odd} and t_{even} that satisfy these equalities, see Section 3).

In words, t_{odd} is the type of lists of channels in which each channel in an odd-indexed position is used for one input, while t_{even} is the type of lists of channel in which each channel in an even-indexed position is used for one input. The reason why this particular decomposition of t_{list} could be interesting is that it enables the sharing of a list containing linear channels among two processes, if we know that one process uses the list according to the type t_{odd} and the other process uses the same list according to the type t_{even} . For

example, the process R defined below

$$P \stackrel{\text{def}}{=} \text{*odd?}(l, acc, r) . \text{case } l \text{ of}$$

$$\quad \text{Nil} \quad \Rightarrow r!acc$$

$$\quad \text{Cons}(x, l') \Rightarrow x?(y) . \text{even!}(l', (acc + y), r)$$

$$Q \stackrel{\text{def}}{=} \text{*even?}(l, acc, r) . \text{case } l \text{ of}$$

$$\quad \text{Nil} \quad \Rightarrow r!acc$$

$$\quad \text{Cons}(x, l') \Rightarrow \text{odd!}(l', acc, r)$$

$$R \stackrel{\text{def}}{=} P \mid Q \mid \text{new } a, b \text{ in } (\text{odd!}(l, 0, a) \mid \text{even!}(l, 0, b) \mid a?(x) . b?(y) . r!(x + y))$$

uses each channel in a list l for receiving a number, sums all such numbers together, and sends the result on another channel r . However, instead of scanning the list l sequentially in a single thread, R spawns two parallel threads (defined by P and Q) that share the *very same list* l : the first thread uses only the odd-indexed channels in l , whereas the second thread uses only the even-indexed channels in l ; the (partial) results obtained by these two threads are collected by R on two locally created channels a and b ; the overall result is eventually sent on r . We are then able to deduce that R makes full use of the channels in l , namely that l has type t_{list} , even though the list as a whole is simultaneously accessed by two parallel threads. In general, we can see that the extension of type combination to composite, potentially infinite types is an effective tool that fosters the parallelization of programs and allows composite data structures containing linear values to be safely shared by a pool of multiple processes, if there is enough information to conclude that each linear value is accessed by exactly one of the processes in the pool.

Such detailed reasoning on the behavior of programs comes at the price of a more sophisticated definition of type combination. This brings us back to the problem of type reconstruction. The reconstruction algorithm described in this article is able to infer the types t_{odd} and t_{even} of the messages accepted by P and Q by looking at the structure of these two processes and of understanding that the overall type of l in R is t_{list} , namely that every channel in l is used exactly once.

Related work. Linear type systems with composite types have been discussed in [8, 9] for the linear π -calculus and in [25] for a functional language. In these works, however, every structure that contains linear values becomes linear itself (there are a few exceptions for specific types [14] or relaxed notions of linearity [11]).

The original type reconstruction algorithm for the linear π -calculus is described in [10]. Our work extends [10] to composite and infinite types. Unlike [10], however, we do not deal with structural subtyping, whose integration into our type reconstruction algorithm is left for future work. The type reconstruction algorithm in [10] and the one we present share a common structure in that they both comprise constraint generation and constraint resolution phases. The main difference concerns the fact that we have to deal with constraints expressing the combination of yet-to-be-determined types, whereas in [10] non-trivial type combinations only apply to channel types. This allows [10] to use an efficient constraint resolution algorithm based on unification. In our setting, the presence of infinite types hinders the use of unification, and in some cases the resolution algorithm may conservatively approximate the outcome in order to ensure proper termination.

P, Q	$::=$	Process
		<code>idle</code> (idle process)
		<code>e?(x).P</code> (input)
		<code>e!f</code> (output)
		<code>P Q</code> (parallel composition)
		<code>*P</code> (process replication)
		<code>new a in P</code> (channel restriction)
		<code>case e {i(x_i) ⇒ P_i}_{i=inl,inr}</code> (pattern matching)
e, f	$::=$	Expression
		<code>n</code> (integer constant)
		<code>u</code> (name)
		<code>(e, f)</code> (pair)
		<code>fst(e)</code> (first projection)
		<code>snd(e)</code> (second projection)
		<code>inl(e)</code> (left injection)
		<code>inr(e)</code> (right injection)

Table 1: Syntax of processes and expressions.

Session types [6, 7] describe *linearized* channels, namely channels that can be used for multiple communications, but only in a sequential way. There is a tight connection between linear and linearized channels: as shown in [13, 4, 3, 2], linearized channels can be encoded in the linear π -calculus. A consequence of this encoding is that the type reconstruction algorithm we present in this article can be used for inferring possibly infinite session types (we will see an example of this feature in Section 7). The task of reconstructing session types directly has been explored in [17], but for finite types only.

Structure of the paper. We present the calculus in Section 2 and the type system in Section 3. The type reconstruction algorithm consists of a constraint generation phase (Section 4) and a constraint resolution phase (Section 5). We discuss some important issues related to the implementation of the algorithm in Section 6 and a few more elaborate examples in Section 7. Section 8 concludes and hints at some ongoing and future work. Proofs of the results in Sections 3 and 4 are in Appendixes A and B, respectively. Appendix C illustrates a few typing derivations of examples discussed in Section 5. A proof-of-concept implementation of the algorithm is available on the author’s home page.

2. THE π -CALCULUS WITH DATA TYPES

In this section we define the syntax and operational semantics of the formal language we work with, which is an extension of the π -calculus featuring base and composite data types and a pattern matching construct.

[S-PAR 1] $\text{idle} \mid P \equiv P$	[S-PAR 2] $P \mid Q \equiv Q \mid P$	[S-PAR 3] $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$	[S-REP] $*P \preceq *P \mid P$
[S-RES 1] $\text{new } a \text{ in new } b \text{ in } P \equiv \text{new } b \text{ in new } a \text{ in } P$		[S-RES 2] $\frac{a \notin \text{fn}(Q)}{(\text{new } a \text{ in } P) \mid Q \equiv \text{new } a \text{ in } (P \mid Q)}$	

Table 2: Structural pre-congruence for processes.

2.1. Syntax. Let us introduce some notation first. We use integer numbers m, n, \dots , a countable set of *channels* a, b, \dots , and a countable set of *variables* x, y, \dots which is disjoint from the set of channels; *names* u, v, \dots are either channels or variables.

The syntax of expressions and processes is given in Table 1. *Expressions* e, f, \dots are either integers, names, pairs (e, f) of expressions, the i -th projection of an expression $i(e)$ where $i \in \{\text{fst}, \text{snd}\}$, or the injection $i(e)$ of an expression e using the constructor $i \in \{\text{inl}, \text{inr}\}$. Using projections **fst** and **snd** instead of a pair splitting construct, as found for instance in [24, 23], is somewhat unconventional, but helps us highlighting some features of our type system. We will discuss some practical aspects of this choice in Section 6.3.

Values v, w, \dots are expressions without variables and occurrences of the projections **fst** and **snd**.

Processes P, Q, \dots comprise and extend the standard constructs of the asynchronous π -calculus. The **idle** process performs no action; the input process $e?(x).P$ waits for a message v from the channel denoted by e and continues as P where x has been replaced by v ; the output process $e!f$ sends the value resulting from the evaluation of f on the channel resulting from the evaluation of e ; the composition $P \mid Q$ executes P and Q in parallel; the replication $*P$ denotes infinitely many copies of P executing in parallel; the restriction **new** a **in** P creates a new channel a with scope P . In addition to these, we include a pattern matching construct **case** $e \{i(x_i) \Rightarrow P_i\}_{i=\text{inl}, \text{inr}}$ which evaluates e to a value of the form $i(v)$ for some $i \in \{\text{inl}, \text{inr}\}$, binds v to x_i and continues as P_i . The notions of *free names* $\text{fn}(P)$ and *bound names* $\text{bn}(P)$ of P are as expected, recalling that **case** $e \{i(x_i) \Rightarrow P_i\}_{i=\text{inl}, \text{inr}}$ binds x_i in P_i . We identify processes modulo renaming of bound names and we write $e\{v/x\}$ and $P\{v/x\}$ for the capture-avoiding substitutions of v for the free occurrences of x in e and P , respectively. Occasionally, we omit **idle** when it is guarded by a prefix.

2.2. Operational semantics. The operational semantics of the language is defined in terms of a structural pre-congruence relation for processes, an evaluation relation for expressions, and a reduction relation for processes. *Structural pre-congruence* \preceq is meant to rearrange process terms which should not be distinguished. The relation is defined in Table 2, where we write $P \equiv Q$ in place of the two inequalities $P \preceq Q$ and $Q \preceq P$. Overall \equiv coincides with the conventional structural congruence of the π -calculus, except that, as in [12], we omit the relation $*P \mid P \preceq *P$ (the reason will be explained in Remark 3.10).

Evaluation $e \downarrow v$ and *reduction* $P \xrightarrow{\ell} Q$ are defined in Table 3. Both relation are fairly standard. As in [15], reduction is decorated with a *label* ℓ that is either a channel or the special symbol τ : in [R-COMM] the label is the channel a on which a message is exchanged; in [R-CASE] it is τ since pattern matching is an internal computation not involving

[E-INT] $n \downarrow n$	[E-CHAN] $a \downarrow a$		
[E-PAIR] $\frac{e_i \downarrow v_i \ (i=1,2)}{(e_1, e_2) \downarrow (v_1, v_2)}$	[E-FST] $\frac{e \downarrow (v, w)}{\mathbf{fst}(e) \downarrow v}$	[E-SND] $\frac{e \downarrow (v, w)}{\mathbf{snd}(e) \downarrow w}$	[E-INR], [E-INL] $\frac{e \downarrow v \quad k \in \{\mathbf{inl}, \mathbf{inr}\}}{k(e) \downarrow k(v)}$
[R-COMM] $\frac{e_i \downarrow a \ (i=1,2) \quad f \downarrow v}{e_1 ! f \mid e_2?(x).Q \xrightarrow{a} Q\{v/x\}}$	[R-CASE] $\frac{e \downarrow k(v) \quad k \in \{\mathbf{inl}, \mathbf{inr}\}}{\mathbf{case} \ e \ \{i(x_i) \Rightarrow P_i\}_{i=\mathbf{inl}, \mathbf{inr}} \xrightarrow{\tau} P_k\{v/x_k\}}$		
[R-PAR] $\frac{P \xrightarrow{\ell} P'}{P \mid Q \xrightarrow{\ell} P' \mid Q}$	[R-NEW 1] $\frac{P \xrightarrow{a} Q}{\mathbf{new} \ a \ \mathbf{in} \ P \xrightarrow{\tau} \mathbf{new} \ a \ \mathbf{in} \ Q}$	[R-NEW 2] $\frac{P \xrightarrow{\ell} Q \quad \ell \neq a}{\mathbf{new} \ a \ \mathbf{in} \ P \xrightarrow{\ell} \mathbf{new} \ a \ \mathbf{in} \ Q}$	
[R-STRUCT] $\frac{P \preceq P' \quad P' \xrightarrow{\ell} Q' \quad Q' \preceq Q}{P \xrightarrow{\ell} Q}$			

Table 3: Evaluation of expressions and reduction of processes.

communications. Note that, as we allow expressions in input and output processes for both the subject and the object of a communication, rule [R-COMM] provides suitable premises to evaluate them. Rules [R-PAR], [R-NEW 1], and [R-NEW 2] propagate labels through parallel compositions and restrictions. In [R-NEW 1], the label a becomes τ when it escapes the scope of a . Rule [R-STRUCT] closes reduction under structural congruence.

Example 2.1 (list sharing). Below are the desugared representations of P and Q discussed in Section 1:

$$\begin{aligned}
 P' &\stackrel{\text{def}}{=} \mathbf{*odd?}(z). \\
 &\quad \mathbf{case} \ \mathbf{fst}(z) \ \mathbf{of} \\
 &\quad \quad \mathbf{inl}(_) \Rightarrow \mathbf{snd}(\mathbf{snd}(z))! \mathbf{fst}(\mathbf{snd}(z)) \\
 &\quad \quad \mathbf{inr}(x) \Rightarrow \mathbf{fst}(x)?(y). \mathbf{even}!(\mathbf{snd}(x), (\mathbf{fst}(\mathbf{snd}(z)) + y, \mathbf{snd}(\mathbf{snd}(z)))) \\
 Q' &\stackrel{\text{def}}{=} \mathbf{*even?}(z). \\
 &\quad \mathbf{case} \ \mathbf{fst}(z) \ \mathbf{of} \\
 &\quad \quad \mathbf{inl}(_) \Rightarrow \mathbf{snd}(\mathbf{snd}(z))! \mathbf{fst}(\mathbf{snd}(z)) \\
 &\quad \quad \mathbf{inr}(x) \Rightarrow \mathbf{odd}!(\mathbf{snd}(x), (\mathbf{fst}(\mathbf{snd}(z)), \mathbf{snd}(\mathbf{snd}(z))))
 \end{aligned}$$

where the constructors \mathbf{inl} and \mathbf{inr} respectively replace \mathbf{Nil} and \mathbf{Cons} , \mathbf{inl} has an (unused) argument denoted by the anonymous variable $_$, and tuple components are accessed using (possibly repeated) applications of \mathbf{fst} and \mathbf{snd} . \blacksquare

3. TYPE SYSTEM

In this section we define a type system for the language presented in Section 2. The type system extends the one for the linear π -calculus [15] with composite and possibly infinite, regular types. The key feature of the linear π -calculus is that channel types are enriched with information about the number of times the channels they denote are used for input/output operations. Such number is abstracted into a *use* κ, \dots , which is an element of the set $\{0, 1, \omega\}$ where 0 and 1 obviously stand for no use and one use only, while ω stands for any number of uses.

Definition 3.1 (types). *Types*, ranged over by t, s, \dots , are the *possibly infinite regular trees* built using the nullary constructor `int`, the unary constructors $[\cdot]^{\kappa_1, \kappa_2}$ for every combination of κ_1 and κ_2 , the binary constructors $\cdot \times \cdot$ (product) and $\cdot \oplus \cdot$ (disjoint sum).

The type $[t]^{\kappa_1, \kappa_2}$ denotes channels for exchanging messages of type t . The uses κ_1 and κ_2 respectively denote how many input and output operations are allowed on the channel. For example: a channel with type $[t]^{0,1}$ cannot be used for input and must be used once for sending a message of type t ; a channel with type $[t]^{0,0}$ cannot be used at all; a channel with type $[t]^{\omega, \omega}$ can be used any number of times for sending and/or receiving messages of type t . A product $t_1 \times t_2$ describes pairs (v_1, v_2) where v_i has type t_i for $i = 1, 2$. A disjoint sum $t_1 \oplus t_2$ describes values of the form `inl`(v) where v has type t_1 or of the form `inr`(v) where v has type t_2 . Throughout the paper we let \odot stand for either \times or \oplus .

We do not provide a concrete, finite syntax for denoting infinite types and work directly with regular trees instead. Recall that a regular tree is a partial function from paths to type constructors (see *e.g.* [22, Chapter 21]), it consists of finitely many distinct subtrees, and admits finite representations using either the well-known μ notation or finite systems of equations [1] (our implementation internally uses both). Working directly with regular trees gives us the coarsest possible notion of type equality ($t = s$ means that t and s are the same partial function) and it allows us to reuse some key results on regular trees that will be essential in the following. In particular, throughout the paper we will implicitly use the next result to define types as solutions of particular systems of equations:

Theorem 3.2. *Let $\{\alpha_i = \mathbb{T}_i \mid 1 \leq i \leq n\}$ be a finite system of equations where each \mathbb{T}_i is a finite term built using the constructors in Definition 3.1 and the pairwise distinct unknowns $\{\alpha_1, \dots, \alpha_n\}$. If none of the \mathbb{T}_i is an unknown, then there exists a unique substitution $\sigma = \{\alpha_i \mapsto t_i \mid 1 \leq i \leq n\}$ such that $t_i = \sigma \mathbb{T}_i$ and t_i is a regular tree for each $1 \leq i \leq n$.*

Proof. All the right hand sides of the equations are finite – hence regular – and different from an unknown, therefore this result is just a particular case of [1, Theorem 4.3.1]. \square

Example 3.3 (integer stream). The type of *integer streams* `int` \times (`int` \times (`int` \times \dots)) is the unique regular tree t such that $t = \text{int} \times t$. To make sense out of this statement we have to be sure that such t does exist and is indeed unique. Consider the equation $\alpha = \text{int} \times \alpha$ obtained from the above equality by turning each occurrence of the metavariable t into the unknown α and observe that the right hand side of such equation is not an unknown. By Theorem 3.2, there exists a unique regular tree t such that $t = \text{int} \times t$. Note that t consists of two distinct subtrees, `int` and t itself. \blacksquare

Example 3.4 (lists). To verify the existence of the types t_{odd} and t_{even} informally introduced in Section 1, consider the system of equations

$$\{\alpha_1 = \text{int} \oplus ([\text{int}]^{1,0} \times \alpha_2), \alpha_2 = \text{int} \oplus ([\text{int}]^{0,0} \times \alpha_1)\}$$

obtained by turning the metavariables t_{odd} and t_{even} in (1.3) respectively into the unknowns α_1 and α_2 and by using basic types and disjoint sums in place of the list constructors **Nil** and **Cons**. Theorem 3.2 says that there exist two unique regular trees t_{odd} and t_{even} such that $t_{odd} = \mathbf{int} \oplus ([\mathbf{int}]^{1,0} \times t_{even})$ and $t_{even} = \mathbf{int} \oplus ([\mathbf{int}]^{0,0} \times t_{odd})$. Similarly, t_{list} is the unique type such that $t_{list} = \mathbf{int} \oplus ([\mathbf{int}]^{1,0} \times t_{list})$. \blacksquare

We now define some key notions on uses and types. To begin with, we define a binary operation $+$ on uses that allows us to express the *combined* use $\kappa_1 + \kappa_2$ of a channel that is used both as denoted by κ_1 and as denoted by κ_2 . Formally:

$$\kappa_1 + \kappa_2 \stackrel{\text{def}}{=} \begin{cases} \kappa_1 & \text{if } \kappa_2 = 0 \\ \kappa_2 & \text{if } \kappa_1 = 0 \\ \omega & \text{otherwise} \end{cases} \quad (3.1)$$

Note that 0 is neutral and ω is absorbing for $+$ and that $1 + 1 = \omega$, since ω is the only use allowing us to express the fact that a channel is used twice. In a few places we will write 2κ as an abbreviation for $\kappa + \kappa$.

We now lift the notion of combination from uses to types. Since types may be infinite, we resort to a coinductive definition.

Definition 3.5 (type combination). Let \mathcal{C}_{type} be the largest relation between pairs of types and types such that $((t_1, t_2), s) \in \mathcal{C}_{type}$ implies either:

- $t_1 = t_2 = s = \mathbf{int}$, or
- $t_1 = [t]^{\kappa_1, \kappa_2}$ and $t_2 = [t]^{\kappa_3, \kappa_4}$ and $s = [t]^{\kappa_1 + \kappa_3, \kappa_2 + \kappa_4}$, or
- $t_1 = t_{11} \odot t_{12}$ and $t_2 = t_{21} \odot t_{22}$ and $s = s_1 \odot s_2$ and $((t_{1i}, t_{2i}), s_i) \in \mathcal{C}_{type}$ for $i = 1, 2$.

Observe that \mathcal{C}_{type} is a partial binary function on types, that is $((t_1, t_2), s_1) \in \mathcal{C}_{type}$ and $((t_1, t_2), s_2) \in \mathcal{C}_{type}$ implies $s_1 = s_2$. When $(t, s) \in \text{dom}(\mathcal{C}_{type})$, we write $t + s$ for $\mathcal{C}_{type}(t, s)$, that is the *combination* of t and s . Occasionally we also write $2t$ in place of $t + t$.

Intuitively, basic types combine with themselves and the combination of channel types with equal message types is obtained by combining corresponding uses. For example, we have $[\mathbf{int}]^{0,1} + [\mathbf{int}]^{1,0} = [\mathbf{int}]^{1,1}$ and $[[\mathbf{int}]^{1,0}]^{0,1} + [[\mathbf{int}]^{1,0}]^{1,1} = [[\mathbf{int}]^{1,0}]^{1,\omega}$. In the latter example, note that the uses of channel types within the top-most ones are *not* combined together. Type combination propagates component-wise on composite types. For instance, we have $([\mathbf{int}]^{0,1} \times [\mathbf{int}]^{0,0}) + ([\mathbf{int}]^{0,0} \times [\mathbf{int}]^{1,0}) = ([\mathbf{int}]^{0,1} + [\mathbf{int}]^{0,0}) \times ([\mathbf{int}]^{0,0} + [\mathbf{int}]^{1,0}) = [\mathbf{int}]^{0,1} \times [\mathbf{int}]^{1,0}$. Unlike use combination, type combination is a partial operation: it is undefined to combine two types having different structures, or to combine two channel types carrying messages of different types. For example, $\mathbf{int} + [\mathbf{int}]^{0,0}$ is undefined and so is $[[\mathbf{int}]^{0,0}]^{0,1} + [[\mathbf{int}]^{0,1}]^{0,1}$, because $[\mathbf{int}]^{0,0}$ and $[\mathbf{int}]^{0,1}$ differ.

Types that can be combined together play a central role, so we name a relation that characterizes them:

Definition 3.6 (coherent types). We say that t and s are *structurally coherent* or simply *coherent*, notation $t \sim s$, if $t + s$ is defined, namely there exists t' such that $((t, s), t') \in \mathcal{C}_{type}$.

Observe that \sim is an equivalence relation, implying that a type can always be combined with itself (*i.e.*, $2t$ is always defined). Type combination is also handy for characterizing a fundamental partitioning of types:

Definition 3.7 (unlimited and linear types). We say that t is *unlimited*, notation $\text{un}(t)$, if $2t = t$. We say that it is *linear* otherwise.

Channel types are either linear or unlimited depending on their uses. For example, $[t]^{0,0}$ is unlimited because $[t]^{0,0} + [t]^{0,0} = [t]^{0,0}$, whereas $[t]^{1,0}$ is linear because $[t]^{1,0} + [t]^{1,0} = [t]^{\omega,0} \neq [t]^{1,0}$. Similarly, $[t]^{\omega,\omega}$ is unlimited while $[t]^{0,1}$ and $[t]^{1,1}$ are linear. Other types are linear or unlimited depending on the channel types occurring in them. For instance, $[t]^{0,0} \times [t]^{1,0}$ is linear while $[t]^{0,0} \times [t]^{\omega,0}$ is unlimited. Note that only the topmost channel types of a type matter. For example, $[[t]^{1,1}]^{0,0}$ is unlimited despite of the fact that it contains the subterm $[t]^{1,1}$ which is itself linear, because such subterm is found within an unlimited channel type.

We use *type environments* to track the type of free names occurring in expressions and processes. Type environments Γ, \dots are finite maps from names to types that we write as $u_1 : t_1, \dots, u_n : t_n$. We identify type environments modulo the order of their associations, write \emptyset for the *empty environment*, $\text{dom}(\Gamma)$ for the *domain* of Γ , namely the set of names for which there is an association in Γ , and Γ_1, Γ_2 for the *union* of Γ_1 and Γ_2 when $\text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2) = \emptyset$. We also extend the partial combination operation $+$ on types to a partial combination operation on type environments, thus:

$$\Gamma_1 + \Gamma_2 \stackrel{\text{def}}{=} \begin{cases} \Gamma_1, \Gamma_2 & \text{if } \text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2) = \emptyset \\ (\Gamma'_1 + \Gamma'_2), u : t_1 + t_2 & \text{if } \Gamma_i = \Gamma'_i, u : t_i \text{ for } i = 1, 2 \end{cases} \quad (3.2)$$

The operation $+$ extends type combination in [15] and the \uplus operator in [24]. Note that $\Gamma_1 + \Gamma_2$ is undefined if there is $u \in \text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2)$ such that $\Gamma_1(u) + \Gamma_2(u)$ is undefined. Note also that $\text{dom}(\Gamma_1 + \Gamma_2) = \text{dom}(\Gamma_1) \cup \text{dom}(\Gamma_2)$. Thinking of type environments as of specifications of the resources used by expressions/processes, $\Gamma_1 + \Gamma_2$ expresses the combined use of the resources specified in Γ_1 and Γ_2 . Any resource occurring in only one of these environments occurs in $\Gamma_1 + \Gamma_2$; any resource occurring in both Γ_1 and Γ_2 is used according to the combination of its types in $\Gamma_1 + \Gamma_2$. For example, if a process sends an integer over a channel a , it will be typed in an environment that contains the association $a : [\text{int}]^{0,1}$; if another process uses the same channel a for receiving an integer, it will be typed in an environment that contains the association $a : [\text{int}]^{1,0}$. Overall, the parallel composition of the two processes uses channel a according to the type $[\text{int}]^{0,1} + [\text{int}]^{1,0} = [\text{int}]^{1,1}$ and therefore it will be typed in an environment that contains the association $a : [\text{int}]^{1,1}$.

The last notion we need before presenting the type rules is that of an unlimited type environment. This is a plain generalization of the notion of unlimited type, extended to the range of a type environment. We say that Γ is unlimited, notation $\text{un}(\Gamma)$, if $\text{un}(\Gamma(u))$ for every $u \in \text{dom}(\Gamma)$. A process typed in an unlimited type environment need not use any of the resources described therein.

Type rules for expressions and processes are presented in Table 4. These rules are basically the same as those found in the literature [15, 10]. The possibility of sharing data structures among several processes, which we have exemplified in Section 1, is a consequence of our notion of type combination extended to composite regular types.

Type rules for expressions are unremarkable. Just observe that unused type environments must be unlimited. Also, the projections **fst** and **snd** discard one component of a pair, so the discarded component must have an unlimited type.

Let us move on to the type rules for processes. The idle process does nothing, so it is well typed only in an unlimited environment. Rule [T-IN] types an input process $e?(x).P$. The subject e must evaluate to a channel whose input use is either 1 or ω and whose output use is either 0 or ω . We capture the first condition saying that the input use of the channel has the form $1 + \kappa_1$ for some κ_1 , and the second condition saying that the output use of

Expressions			
$\frac{[T\text{-INT}]}{\text{un}(\Gamma)} \Gamma \vdash n : \mathbf{int}$	$\frac{[T\text{-NAME}]}{\text{un}(\Gamma)} \Gamma, u : t \vdash u : t$	$\frac{[T\text{-INL}]}{\Gamma \vdash \mathbf{inl}(e) : t \oplus s}$	$\frac{[T\text{-INR}]}{\Gamma \vdash \mathbf{inr}(e) : t \oplus s}$
$\frac{[T\text{-PAIR}]}{\Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : t_1 \times t_2} \Gamma_i \vdash e_i : t_i \ (i=1,2)$	$\frac{[T\text{-FST}]}{\Gamma \vdash \mathbf{fst}(e) : t} \Gamma \vdash e : t \times s \quad \text{un}(s)$	$\frac{[T\text{-SND}]}{\Gamma \vdash \mathbf{snd}(e) : s} \Gamma \vdash e : t \times s \quad \text{un}(t)$	
Processes			
$\frac{[T\text{-IDLE}]}{\Gamma \vdash \mathbf{idle}} \text{un}(\Gamma)$	$\frac{[T\text{-IN}]}{\Gamma_1 + \Gamma_2 \vdash e?(x).P} \Gamma_1 \vdash e : [t]^{1+\kappa_1, 2\kappa_2} \quad \Gamma_2, x : t \vdash P$	$\frac{[T\text{-OUT}]}{\Gamma_1 + \Gamma_2 \vdash e!f} \Gamma_1 \vdash e : [t]^{2\kappa_1, 1+\kappa_2} \quad \Gamma_2 \vdash f : t$	
$\frac{[T\text{-REP}]}{\Gamma \vdash *P} \Gamma \vdash P \quad \text{un}(\Gamma)$	$\frac{[T\text{-PAR}]}{\Gamma_1 + \Gamma_2 \vdash P_1 \mid P_2} \Gamma_i \vdash P_i \ (i=1,2)$	$\frac{[T\text{-NEW}]}{\Gamma \vdash \mathbf{new} \ a \ \mathbf{in} \ P} \Gamma, a : [t]^{\kappa, \kappa} \vdash P$	
$\frac{[T\text{-CASE}]}{\Gamma_1 + \Gamma_2 \vdash \mathbf{case} \ e \ \{i(x_i) \Rightarrow P_i\}_{i=\mathbf{inl}, \mathbf{inr}}} \Gamma_1 \vdash e : t \oplus s \quad \Gamma_2, x_i : t \vdash P_i \ (i=\mathbf{inl}, \mathbf{inr})$			

Table 4: Type rules for expressions and processes.

the channel has the form $2\kappa_2$ for some κ_2 . The continuation P is typed in an environment enriched with the association for the received message x . Note the combination $\Gamma_1 + \Gamma_2$ in the conclusion of rule [T-IN]. In particular, if e evaluates to a linear channel, its input capability is consumed by the operation and such channel can no longer be used for inputs in the continuation. Rule [T-OUT] types an output process $e!f$. The rule is dual to [T-IN] in that it requires the channel to which e evaluates to have a positive output use. Rule [T-REP] states that a replicated process $*P$ is well typed in the environment Γ provided that P is well typed in an unlimited Γ . The rationale is that $*P$ stands for an unbounded number of copies of P composed in parallel, hence P cannot contain (free) linear channels. The rules [T-PAR] and [T-CASE] are conventional, with the by now familiar use of environment combination for properly distributing linear resources to the various subterms of a process. The rule [T-NEW] is also conventional. We require the restricted channel to have the same input and output uses. While this is not necessary for the soundness of the type system, in practice it is a reasonable requirement. We also argue that this condition is important for the modular application of the type reconstruction algorithm; we will discuss this aspect more in detail in Section 6.

As in many behavioral type systems, the type environment in which the reducing process is typed may change as a consequence of the reduction. More specifically, reductions involving a communication on channels *consume* 1 unit from both the input and output uses

of the channel's type. In order to properly state subject reduction, we define a reduction relation over type environments. In particular, we write $\xrightarrow{\ell}$ for the least relation between type environments such that

$$\Gamma \xrightarrow{\tau} \Gamma \quad \Gamma + a : [t]^{1,1} \xrightarrow{a} \Gamma$$

In words, $\xrightarrow{\tau}$ denotes an internal computation (pattern matching) or a communication on some restricted channel which does not consume any resource from the type environment, while \xrightarrow{a} denotes a communication on channel a which consumes 1 use from both the input and output slots in a 's type. For example, we have

$$a : [\mathbf{int}]^{1,1} \xrightarrow{a} a : [\mathbf{int}]^{0,0}$$

by taking $\Gamma \stackrel{\text{def}}{=} a : [\mathbf{int}]^{0,0}$ in the definition of \xrightarrow{a} above, since $\Gamma + a : [\mathbf{int}]^{1,1} = a : [\mathbf{int}]^{1,1}$. The residual environment denotes the fact that the (linear) channel a can no longer be used for communication.

Now we have:

Theorem 3.8. *Let $\Gamma \vdash P$ and $P \xrightarrow{\ell} Q$. Then $\Gamma' \vdash Q$ for some Γ' such that $\Gamma \xrightarrow{\ell} \Gamma'$.*

Theorem 3.8 establishes not only a subject reduction result, but also a soundness result because it implies that a channel is used no more than its type allows. It is possible to establish more properties of the linear π -calculus, such as the fact that communications involving linear channels enjoy partial confluence. In this work we focus on the issue of type reconstruction. The interested reader may refer to [15] for further results.

Example 3.9. We consider again the processes P' and Q' in Example 2.1 and sketch a few key derivation steps to argue that they are well typed. To this aim, consider the types t_{odd} , t_{even} , and t_{zero} that satisfy the equalities below

$$\begin{aligned} t_{\text{odd}} &= \mathbf{int} \oplus ([\mathbf{int}]^{0,1} \times t_{\text{even}}) \\ t_{\text{even}} &= \mathbf{int} \oplus ([\mathbf{int}]^{0,0} \times t_{\text{odd}}) \\ t_{\text{zero}} &= \mathbf{int} \oplus ([\mathbf{int}]^{0,0} \times t_{\text{zero}}) \end{aligned}$$

and also consider the types of the messages respectively carried by **odd** and **even**:

$$\begin{aligned} s_{\text{odd}} &\stackrel{\text{def}}{=} t_{\text{odd}} \times (\mathbf{int} \times [\mathbf{int}]^{0,1}) \\ s_{\text{even}} &\stackrel{\text{def}}{=} t_{\text{even}} \times (\mathbf{int} \times [\mathbf{int}]^{0,1}) \end{aligned}$$

Now, in the **inl** branch of P' we derive (D1)

$$\frac{\frac{\frac{z : t_{\text{zero}} \times (\mathbf{int} \times [\mathbf{int}]^{0,1}) \vdash z : t_{\text{zero}} \times (\mathbf{int} \times [\mathbf{int}]^{0,1})}{\text{[T-NAME]}}}{z : t_{\text{zero}} \times (\mathbf{int} \times [\mathbf{int}]^{0,1}) \vdash \mathbf{snd}(z) : \mathbf{int} \times [\mathbf{int}]^{0,1}}{\text{[T-SND]}}}{z : t_{\text{zero}} \times (\mathbf{int} \times [\mathbf{int}]^{0,1}) \vdash \mathbf{snd}(\mathbf{snd}(z)) : [\mathbf{int}]^{0,1}}{\text{[T-SND]}}$$

using the fact that $\text{un}(t_{\text{zero}})$ and $\text{un}(\mathbf{int})$. We also derive (D2)

$$\frac{\frac{\frac{z : t_{\text{zero}} \times (\mathbf{int} \times [\mathbf{int}]^{0,0}) \vdash z : t_{\text{zero}} \times (\mathbf{int} \times [\mathbf{int}]^{0,0})}{\text{[T-NAME]}}}{z : t_{\text{zero}} \times (\mathbf{int} \times [\mathbf{int}]^{0,0}) \vdash \mathbf{snd}(z) : \mathbf{int} \times [\mathbf{int}]^{0,0}}{\text{[T-SND]}}}{z : t_{\text{zero}} \times (\mathbf{int} \times [\mathbf{int}]^{0,0}) \vdash \mathbf{fst}(\mathbf{snd}(z)) : \mathbf{int}}{\text{[T-FST]}}$$

using the fact that $\text{un}(t_{zero})$ and $\text{un}([\mathbf{int}]^{0,0})$, therefore we derive (D3)

$$\frac{\text{(D1)} \quad \text{(D2)}}{z : t_{zero} \times (\mathbf{int} \times [\mathbf{int}]^{0,1}), _ : \mathbf{int} \vdash \mathbf{snd}(\mathbf{snd}(z)) \mathbf{!fst}(\mathbf{snd}(z))} \text{[T-OUT]}$$

using the combination

$$(t_{zero} \times (\mathbf{int} \times [\mathbf{int}]^{0,1})) + (t_{zero} \times (\mathbf{int} \times [\mathbf{int}]^{0,0})) = t_{zero} \times (\mathbf{int} \times [\mathbf{int}]^{0,1})$$

Already in this sub-derivation we appreciate that although the pair z is accessed twice, its type in the conclusion of (D3) correctly tracks the fact that the channel contained in z is only used once, for an output.

For the \mathbf{inr} branch in P' there exists another derivation (D4) concluding

$$\frac{\vdots \quad \vdots}{\mathbf{even} : [s_{even}]^{0,\omega}, x : [\mathbf{int}]^{1,0} \times t_{even}, z : t_{zero} \times \mathbf{int} \times [\mathbf{int}]^{1,0} \vdash \mathbf{fst}(x)?(y) \dots} \text{[T-IN]}$$

Now we conclude

$$\frac{\frac{\frac{\text{[T-NAME]}}{z : s_{odd} \vdash z : s_{odd}} \quad \text{(D3)} \quad \text{(D4)}}{\mathbf{even} : [s_{even}]^{0,\omega}, z : s_{odd} \vdash \mathbf{case} \ z \ \mathbf{of} \ \dots} \text{[T-CASE]}}{\mathbf{odd} : [s_{odd}]^{\omega,0}, \mathbf{even} : [s_{even}]^{0,\omega} \vdash \mathbf{odd}?(z) \ . \ \mathbf{case} \ z \ \mathbf{of} \ \dots} \text{[T-IN]}}{\mathbf{odd} : [s_{odd}]^{\omega,0}, \mathbf{even} : [s_{even}]^{0,\omega} \vdash P'} \text{[T-REP]}$$

Note that \mathbf{odd} and \mathbf{even} must be unlimited channels because they occur free in a replicated process, for which rule [T-REP] requires an unlimited environment. A similar derivation shows that Q' is well typed in an environment where the types of \mathbf{odd} and \mathbf{even} have swapped uses

$$\frac{\vdots}{\mathbf{odd} : [s_{odd}]^{0,\omega}, \mathbf{even} : [s_{even}]^{\omega,0} \vdash Q'} \text{[T-REP]}$$

so the combined types of \mathbf{odd} and \mathbf{even} are $[s_{odd}]^{\omega,\omega}$ and $[s_{even}]^{\omega,\omega}$, respectively. Using these, we find a typing derivation for the process R in Section 1. Proceeding bottom-up we have

$$\frac{\vdots}{\mathbf{odd} : [s_{odd}]^{\omega,\omega}, l : t_{odd}, a : [\mathbf{int}]^{0,1} \vdash \mathbf{odd}!(l,0,a)} \text{[T-OUT]}$$

and

$$\frac{\vdots}{\mathbf{even} : [s_{even}]^{\omega,\omega}, l : t_{even}, b : [\mathbf{int}]^{0,1} \vdash \mathbf{even}!(l,0,b)} \text{[T-OUT]}$$

as well as

$$\frac{\vdots}{a : [\mathbf{int}]^{1,0}, b : [\mathbf{int}]^{1,0}, r : [\mathbf{int}]^{0,1} \vdash a?(x) \ . b?(y) \ . r!(x+y)} \text{[T-IN]}$$

from which we conclude

$$\frac{\vdots}{\mathbf{odd} : [s_{odd}]^{\omega,\omega}, \mathbf{even} : [s_{even}]^{\omega,\omega}, l : t_{list}, r : [\mathbf{int}]^{0,1} \vdash \mathbf{new} \ a, b \ \mathbf{in} \ \dots} \text{[T-NEW]} \text{ (twice)}$$

using the property $t_{odd} + t_{even} = t_{list}$. ■

We conclude this section with a technical remark to justify the use of a structural precongruence relation in place of a more familiar symmetric one.

Remark 3.10. Let us show why the relation $*P \mid P \preceq *P$ would invalidate Theorem 3.8 (more specifically, Lemma A.3) in our setting (a similar phenomenon is described in [12]). To this aim, consider the process

$$P \stackrel{\text{def}}{=} a?(x).\mathbf{new} \ c \ \mathbf{in} \ (*c?(y).c!y \mid c!b)$$

and the type environment $\Gamma_\kappa \stackrel{\text{def}}{=} a : [\mathbf{int}]^{\omega,0}, b : [\mathbf{int}]^{0,\kappa}$ for an arbitrary κ . We can derive

$$\begin{array}{c} \vdots \quad \vdots \\ \frac{\frac{\frac{\vdots \quad \vdots}{c : [[\mathbf{int}]^{0,\kappa}]^{\omega,\omega}, y : [\mathbf{int}]^{0,\kappa} \vdash c!y}{} [\text{T-OUT}]}{c : [[\mathbf{int}]^{0,\kappa}]^{\omega,\omega} \vdash c?(y).c!y} [\text{T-IN}]}{c : [[\mathbf{int}]^{0,\kappa}]^{\omega,\omega} \vdash *c?(y).c!y} [\text{T-REP}] \quad \frac{\frac{\vdots \quad \vdots}{\Gamma_\kappa, c : [[\mathbf{int}]^{0,\kappa}]^{0,1} \vdash c!b}{} [\text{T-OUT}]}{\Gamma_\kappa, c : [[\mathbf{int}]^{0,\kappa}]^{\omega,\omega} \vdash *c?(y).c!y \mid c!b} [\text{T-PAR}]}{\Gamma_\kappa, c : [[\mathbf{int}]^{0,\kappa}]^{\omega,\omega} \vdash *c?(y).c!y \mid c!b} [\text{T-NEW}]}{\Gamma_\kappa \vdash \mathbf{new} \ c \ \mathbf{in} \ \dots} [\text{T-IN}]}{\Gamma_\kappa \vdash P} [\text{T-IN}] \end{array}$$

where we have elided a few obvious typing derivations for expressions. In particular, we can find a derivation where b has an unlimited type ($\kappa = 0$) and another one where b has a linear type ($\kappa = 1$). This is possible because channel c , which is restricted within P , can be given different types – respectively, $[[\mathbf{int}]^{0,0}]^{\omega,\omega}$ and $[[\mathbf{int}]^{0,1}]^{\omega,\omega}$ – in the two derivations. We can now obtain

$$\frac{\frac{\frac{\vdots}{\Gamma_0 \vdash P}{} [\text{T-REP}]}{\Gamma_0 \vdash *P} \quad \frac{\frac{\vdots}{\Gamma_1 \vdash P}{} [\text{T-REP}]}{\Gamma_1 \vdash *P \mid P} [\text{R-PAR}]}{\Gamma_1 \vdash *P \mid P}$$

because $\text{un}(\Gamma_0)$ and $\Gamma_0 + \Gamma_1 = \Gamma_1$. If we allowed the structural congruence rule $*P \mid P \preceq *P$, then $\Gamma_1 \vdash *P$ would *not* be derivable because Γ_1 is linear, hence typing would not be preserved by structural pre-congruence. This problem is avoided in [15, 10] by limiting replication to input prefixes, omitting any structural congruence rule for replications, and adding a dedicated synchronization rule for them. In [15] it is stated that ‘‘the full pi-calculus replication operator poses no problems for the linear type system’’, but this holds because there the calculus is typed, so multiple typing derivations for the same process P above would assign the same type to c and, in turn, the same type to b . ■

4. CONSTRAINT GENERATION

We formalize the problem of type reconstruction as follows: given a process P , find a type environment Γ such that $\Gamma \vdash P$, provided there is one. In general, in the derivation for $\Gamma \vdash P$ we also want to identify as many linear channels as possible. We will address this latter aspect in Section 5.

$T, S ::=$	Type expression	$U, V ::=$	Use expression
	α (type variable)		ϱ (use variable)
	int (integer)		κ (use constant)
	$[T]^{U, V}$ (channel)		$U + V$ (use combination)
	$T \times S$ (product)		
	$T \oplus S$ (disjoint sum)		

Table 5: Syntax of use and type expressions.

4.1. Syntax-directed generation algorithm. The type rules shown in Table 4 rely on a fair amount of guessing that concerns the structure of types in the type environment, how they are split/combined using $+$, and the uses occurring in them. So, these rules cannot be easily interpreted as a type reconstruction algorithm. The way we follow to define one is conventional: first, we give an alternative set of (almost) syntax-directed rules that generate *constraints* on types; then, we search for a solution of such constraints. The main technical challenge is that we cannot base our type reconstruction algorithm on conventional unification because we have to deal with constraints expressing not only the *equality* between types and uses, but also the *combination* of types and uses. In addition, we work with possibly infinite types.

To get started, we introduce use and type *expressions*, which share the same structure as uses/types but they differ from them in two fundamental ways:

- (1) We allow use/type variables to stand for unknown uses/types.
- (2) We can express symbolically the combination of use expressions.

We therefore introduce a countable set of *use variables* ϱ, \dots as well as a countable set of *type variables* α, β, \dots ; the syntax of use expressions U, V, \dots and of type expressions T, S, \dots is given in Table 5. Observe that every use is also a use expression and every finite type is also a type expression. We say that T is *proper* if it is different from a type variable.

Constraints φ, \dots are defined by the grammar below:

$\varphi ::=$	Constraint
	$T \hat{=} S$ (type equality)
	$T \hat{=} S_1 + S_2$ (type combination)
	$T \sim S$ (type coherence)
	$U \hat{=} V$ (use equality)

Constraints express relations between types/uses that must be satisfied in order for a given process to be well typed. In particular, we need to express equality constraints between types ($T \hat{=} S$) and uses ($U \hat{=} V$), coherence constraints ($T \sim S$), and combination constraints between types ($T \hat{=} S_1 + S_2$). We will write $\text{un}(T)$ as an abbreviation for the constraint $T \hat{=} T + T$. This notation is motivated by Definition 3.7, according to which a type is unlimited if and only if it is equal to its own combination. We let \mathcal{C}, \dots range over finite constraint sets. The set of *expressions* of a constraint set \mathcal{C} , written $\text{expr}(\mathcal{C})$, is the (finite) set of use and type expressions occurring in the constraints in \mathcal{C} .

The type reconstruction algorithm *generates* type environments for the expressions and processes being analyzed. Unlike the environments in Section 3, these environments associate names with type expressions. For this reason we will let Δ, \dots range over the

$\frac{[\text{C-ENV 1}] \quad \text{dom}(\Delta_1) \cap \text{dom}(\Delta_2) = \emptyset}{\Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta_1, \Delta_2; \emptyset}$	$\frac{[\text{C-ENV 2}] \quad \Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \mathcal{C} \quad \alpha \text{ fresh}}{(\Delta_1, u : \mathsf{T}) \sqcup (\Delta_2, u : \mathsf{S}) \rightsquigarrow \Delta, u : \alpha; \mathcal{C} \cup \{\alpha \hat{=} \mathsf{T} + \mathsf{S}\}}$
$\frac{[\text{M-ENV 1}] \quad \emptyset \sqcap \emptyset \rightsquigarrow \emptyset; \emptyset}{\emptyset \sqcap \emptyset \rightsquigarrow \emptyset; \emptyset}$	$\frac{[\text{M-ENV 2}] \quad \Delta_1 \sqcap \Delta_2 \rightsquigarrow \Delta; \mathcal{C}}{(\Delta_1, u : \mathsf{T}) \sqcap (\Delta_2, u : \mathsf{S}) \rightsquigarrow \Delta, u : \mathsf{T}; \mathcal{C} \cup \{\mathsf{T} \hat{=} \mathsf{S}\}}$

Table 6: Combining and merging operators for type environments.

environments generated by the reconstruction algorithm, although we will refer to them as type environments.

The algorithm also uses two auxiliary operators \sqcup and \sqcap defined in Table 6. The relation $\Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \mathcal{C}$ combines two type environments Δ_1 and Δ_2 into Δ when the names in $\text{dom}(\Delta_1) \cup \text{dom}(\Delta_2)$ are used *both* as specified in Δ_1 *and also* as specified in Δ_2 and, in doing so, generates a set of constraints \mathcal{C} . So \sqcup is analogous to $+$ in (3.2). When Δ_1 and Δ_2 have disjoint domains, Δ is just the union of Δ_1 and Δ_2 and no constraints are generated. Any name u that occurs in $\text{dom}(\Delta_1) \cap \text{dom}(\Delta_2)$ is used according to the combination of $\Delta_1(u)$ and $\Delta_2(u)$. In general, $\Delta_1(u)$ and $\Delta_2(u)$ are type expressions with free type variables, hence this combination cannot be “computed” or “checked” right away. Instead, it is recorded as the constraint $\alpha \hat{=} \Delta_1(u) + \Delta_2(u)$ where α is a fresh type variable.

The relation $\Delta_1 \sqcap \Delta_2 \rightsquigarrow \Delta; \mathcal{C}$ merges two type environments Δ_1 and Δ_2 into Δ when the names in $\text{dom}(\Delta_1) \cup \text{dom}(\Delta_2)$ are used *either* as specified in Δ_1 *or* as specified in Δ_2 and, in doing so, generates a constraint set \mathcal{C} . This merging is necessary when typing the alternative branches of a **case**: recall that rule [T-CASE] in Table 4 requires the *same* type environment Γ for typing the two branches of a **case**. Consequently, $\Delta_1 \sqcap \Delta_2$ is defined only when Δ_1 and Δ_2 have the same domain, and produces a set of constraints \mathcal{C} saying that the corresponding types of the names in Δ_1 and Δ_2 must be equal.

The rules of the type reconstruction algorithm are presented in Table 7 and derive judgments $e : \mathsf{T} \blacktriangleright \Delta; \mathcal{C}$ for expressions and $P \blacktriangleright \Delta; \mathcal{C}$ for processes. In both cases, Δ is the generated environment that contains associations for all the free names in e and P , while \mathcal{C} is the set of constraints that must hold in order for e or P to be well typed in Δ . In a judgment $e : \mathsf{T} \blacktriangleright \Delta; \mathcal{C}$, the type expression T denotes the type of the expression e .

There is a close correspondence between the type system (Table 4) and the reconstruction algorithm (Table 7). In a nutshell, unknown uses and types become fresh use and type variables (all use/type variables introduced by the rules are assumed to be fresh), every application of $+$ in Table 4 becomes an application of \sqcup in Table 7, and every assumption on the form of types becomes a constraint. Constraints accumulate from the premises to the conclusion of each rule of the reconstruction algorithm, which we now review briefly.

Rule [I-INT] deals with integer constants. Their type is obviously **int**, they contain no free names and therefore they generate the empty environment and the empty set of constraints. Rule [I-NAME] deals with the free occurrence of a name u . A fresh type variable standing for the type of *this occurrence* of u is created and used in the resulting type environment $u : \alpha$. Again, no constraints are generated. In general, different occurrences of the same name may have different types which are eventually combined with α later on in the reconstruction process. In rules [I-INL] and [I-INR] the type of the summand that

Expressions	
$\frac{[I\text{-INT}]}{n : \mathbf{int} \triangleright \emptyset; \emptyset}$	$\frac{[I\text{-NAME}]}{u : \alpha \triangleright u : \alpha; \emptyset}$
$\frac{[I\text{-INL}]}{e : \mathsf{T} \triangleright \Delta; \mathcal{C}} \quad \frac{[I\text{-INR}]}{e : \mathsf{T} \triangleright \Delta; \mathcal{C}}$	
$\frac{\mathbf{inl}(e) : \mathsf{T} \oplus \alpha \triangleright \Delta; \mathcal{C}}{\quad} \quad \frac{\mathbf{inr}(e) : \alpha \oplus \mathsf{T} \triangleright \Delta; \mathcal{C}}{\quad}$	
$\frac{[I\text{-PAIR}]}{e_i : \mathsf{T}_i \triangleright \Delta_i; \mathcal{C}_i \quad \Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \mathcal{C}_3 \quad (i=1,2)}{(e_1, e_2) : \mathsf{T}_1 \times \mathsf{T}_2 \triangleright \Delta; \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$	
$\frac{[I\text{-FST}]}{e : \mathsf{T} \triangleright \Delta; \mathcal{C}}$	$\frac{[I\text{-SND}]}{e : \mathsf{T} \triangleright \Delta; \mathcal{C}}$
$\frac{\mathbf{fst}(e) : \alpha \triangleright \Delta; \mathcal{C} \cup \{\mathsf{T} \hat{=} \alpha \times \beta, \mathbf{un}(\beta)\}}{\quad} \quad \frac{\mathbf{snd}(e) : \beta \triangleright \Delta; \mathcal{C} \cup \{\mathsf{T} \hat{=} \alpha \times \beta, \mathbf{un}(\alpha)\}}{\quad}$	
Processes	
$\frac{[I\text{-IDLE}]}{\mathbf{idle} \triangleright \emptyset; \emptyset}$	$\frac{[I\text{-IN}]}{e : \mathsf{T} \triangleright \Delta_1; \mathcal{C}_1 \quad P \triangleright \Delta_2, x : \mathsf{S}; \mathcal{C}_2 \quad \Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \mathcal{C}_3}{e?(x).P \triangleright \Delta; \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\mathsf{T} \hat{=} [\mathsf{S}]^{1+\varrho_1, 2\varrho_2}\}}$
$\frac{[I\text{-OUT}]}{e : \mathsf{T} \triangleright \Delta_1; \mathcal{C}_1 \quad f : \mathsf{S} \triangleright \Delta_2; \mathcal{C}_2 \quad \Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \mathcal{C}_3}{e!f \triangleright \Delta; \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\mathsf{T} \hat{=} [\mathsf{S}]^{2\varrho_1, 1+\varrho_2}\}}$	$\frac{[I\text{-REP}]}{P \triangleright \Delta; \mathcal{C} \quad \Delta \sqcup \Delta \rightsquigarrow \Delta'; \mathcal{C}'}{*P \triangleright \Delta'; \mathcal{C} \cup \mathcal{C}'}$
$\frac{[I\text{-PAR}]}{P_i \triangleright \Delta_i; \mathcal{C}_i \quad \Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \mathcal{C}_3 \quad (i=1,2)}{P_1 \mid P_2 \triangleright \Delta; \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$	$\frac{[I\text{-NEW}]}{P \triangleright \Delta, a : \mathsf{T}; \mathcal{C}}{\mathbf{new} \ a \ \mathbf{in} \ P \triangleright \Delta; \mathcal{C} \cup \{\mathsf{T} \hat{=} [\alpha]^{e, \varrho}\}}$
$\frac{[I\text{-CASE}]}{P_i \triangleright \Delta_i, x_i : \mathsf{T}_i; \mathcal{C}_i \quad e : \mathsf{T} \triangleright \Delta_1; \mathcal{C}_1 \quad \Delta_{\mathbf{inl}} \sqcap \Delta_{\mathbf{inr}} \rightsquigarrow \Delta_2; \mathcal{C}_2 \quad \Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \mathcal{C}_3 \quad (i=\mathbf{inl}, \mathbf{inr})}{\mathbf{case} \ e \ \{i(x_i) \Rightarrow P_i\}_{i=\mathbf{inl}, \mathbf{inr}} \triangleright \Delta; \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \mathcal{C}_{\mathbf{inl}} \cup \mathcal{C}_{\mathbf{inr}} \cup \{\mathsf{T} \hat{=} \mathsf{T}_{\mathbf{inl}} \oplus \mathsf{T}_{\mathbf{inr}}\}}$	
$\frac{[I\text{-WEAK}]}{P \triangleright \Delta; \mathcal{C}}{P \triangleright \Delta, u : \alpha; \mathcal{C} \cup \{\mathbf{un}(\alpha)\}}$	

Table 7: Constraint generation for expressions and processes.

was guessed in [T-INL] and [T-INR] becomes a fresh type variable. Rule [T-PAIR] creates a product type from the type of the components of the pairs, combines the corresponding environments and joins all the constraints generated in the process. Rules [I-FST] and [I-SND] deal with pair projections. The type T of the projected expression must be a product of the form $\alpha \times \beta$. Since the first projection discards the second component of a pair, β must be unlimited in [I-FST]. Symmetrically for [I-SND].

Continuing on with the rules for processes, let us consider [I-IN] and [I-OUT]. The main difference between these rules and the corresponding ones [T-IN] and [T-OUT] is that the use information of the channel on which the communication occurs is unknown, hence it is represented using fresh use variables. The $1 + \varrho_i$ part accounts for the fact that the channel is being used at least once, for an input or an output. The $2\varrho_j$ part accounts for the fact that the use information concerning the capability (either input or output) that is not exercised must be unlimited (note that we extend the notation 2κ to use expressions). Rule [I-REP] deals with a replicated process $*P$. In the type system, $*P$ is well typed in an unlimited environment. Here, we are building up the type environment for $*P$ and we do so by combining the environment Δ generated by P with itself. The rationale is that $\Delta \sqcup \Delta$ yields an unlimited type environment that grants at least all the capabilities granted by Δ . By now most of the main ingredients of the constraint generation algorithm have been revealed, and the remaining rules contain no further novelties but the expected use of the merging operator \sqcap in [I-CASE]. There is, however, a rule [I-WEAK] that has no correspondence in Table 4. This rule is necessary because [I-IN], [I-NEW], and [I-CASE], which correspond to the binding constructs of the calculus, *assume* that the names they bind do occur in the premises on these rules. But since type environments are generated by the algorithm as it works through an expression or a process, this may not be the case if a bound name is never used and therefore never occurs in that expression or process. Furthermore, the \sqcap operator is defined only on type environments having the same domain. This may not be the case if a name occurs in only one branch of a pattern matching, and not in the other one. With rule [I-WEAK] we can introduce missing names in type environments wherever this is necessary. Naturally, an unused name has an unknown type α that must be unlimited, whence the constraint $\text{un}(\alpha)$ (see Example 4.4 for an instance where [I-WEAK] is necessary). Strictly speaking, with [I-WEAK] this set of rules is not syntax directed, which in principle is a problem if we want to consider this as an algorithm. In practice, the places where [I-WEAK] may be necessary are easy to spot (in the premises of all the aforementioned rules for the binding constructs). What we gain with [I-WEAK] is a simpler presentation of the rules for constraint generation.

4.2. Correctness and completeness. If the constraint set generated from P is satisfiable, then it corresponds to a typing for P . To formalize this property, we must first define what “satisfiability” means for a constraint set.

A *substitution* σ is a finite map from type variables to types and from use variables to uses. We write $\text{dom}(\sigma)$ for the set of type and use variables for which there is an association in σ . The *application* of a substitution σ to a use/type expression U/T , respectively denoted by σU and σT , replaces use variables ϱ and type variables α in U/T with the corresponding uses $\sigma(\varrho)$ and types $\sigma(\alpha)$ and computes use combinations whenever possible:

$$\sigma U \stackrel{\text{def}}{=} \begin{cases} \sigma(\varrho) & \text{if } U = \varrho \in \text{dom}(\sigma) \\ \sigma U_1 + \sigma U_2 & \text{if } U = U_1 + U_2 \\ U & \text{otherwise} \end{cases} \quad \sigma T \stackrel{\text{def}}{=} \begin{cases} \sigma(\alpha) & \text{if } T = \alpha \in \text{dom}(\sigma) \\ [\sigma S]^{\sigma U, \sigma V} & \text{if } T = [S]^{U, V} \\ \sigma T_1 \odot \sigma T_2 & \text{if } T = T_1 \odot T_2 \\ T & \text{otherwise} \end{cases}$$

We will make sure that the application of a substitution σ to a type expression T is always well defined: either $\text{dom}(\sigma)$ contains no type variables, in which case σT is a type expression, or $\text{dom}(\sigma)$ includes all use/type variables occurring in T , in which case we say that σ *covers*

\mathbb{T} and $\sigma\mathbb{T}$ is a type. We extend application pointwise to type environments, namely $\sigma\Delta \stackrel{\text{def}}{=} \{u : \sigma\Delta(u) \mid u \in \text{dom}(\Delta)\}$, and we say that σ *covers* Δ if it covers all the type expressions in the range of Δ .

Definition 4.1 (solution, satisfiability, equivalence). A substitution σ is a *solution* for a constraint set \mathcal{C} if it covers all the $\mathbb{T} \in \text{expr}(\mathcal{C})$ and the following conditions hold:

- $\mathbb{T} \hat{=} \mathbb{S} \in \mathcal{C}$ implies $\sigma\mathbb{T} = \sigma\mathbb{S}$, and
- $\mathbb{T} \hat{=} \mathbb{S}_1 + \mathbb{S}_2 \in \mathcal{C}$ implies $\sigma\mathbb{T} = \sigma\mathbb{S}_1 + \sigma\mathbb{S}_2$, and
- $\mathbb{T} \hat{\sim} \mathbb{S} \in \mathcal{C}$ implies $\sigma\mathbb{T} \sim \sigma\mathbb{S}$, and
- $\mathbb{U} \hat{=} \mathbb{V} \in \mathcal{C}$ implies $\sigma\mathbb{U} = \sigma\mathbb{V}$.

We say that \mathcal{C} is *satisfiable* if it has a solution and *unsatisfiable* otherwise. We say that \mathcal{C}_1 and \mathcal{C}_2 are *equivalent* if they have the same solutions.

We can now state the correctness result for the type reconstruction algorithm:

Theorem 4.2. *If $P \blacktriangleright \Delta; \mathcal{C}$ and σ is a solution for \mathcal{C} that covers Δ , then $\sigma\Delta \vdash P$.*

Note that Theorem 4.2 not only requires σ to be a solution for \mathcal{C} , but also that σ must include suitable substitutions for all use and type variables occurring in Δ . Indeed, it may happen that Δ contains use/type variables not involved in any constraint in \mathcal{C} , therefore a solution for \mathcal{C} does not necessarily cover Δ .

The reconstruction algorithm is also complete, in the sense that *each* type environment Γ such that $\Gamma \vdash P$ can be obtained by applying a solution for \mathcal{C} to Δ .

Theorem 4.3. *If $\Gamma \vdash P$, then there exist Δ, \mathcal{C} , and σ such that $P \blacktriangleright \Delta; \mathcal{C}$ and σ is a solution for \mathcal{C} that covers Δ and $\Gamma = \sigma\Delta$.*

Example 4.4. Below we illustrate the reconstruction algorithm at work on the process

`new a in (a!3 | a?(x).idle)`

which will be instrumental also in the following section:

$$\begin{array}{c}
\frac{\frac{\frac{}{a : \alpha_1 \blacktriangleright a : \alpha_1; \emptyset} \quad \frac{}{3 : \text{int} \blacktriangleright \emptyset; \emptyset}}{a!3 \blacktriangleright a : \alpha_1; \{\alpha_1 \hat{=} [\text{int}]^{2\varrho_1, 1+\varrho_2}\}} \text{[I-OUT]} \quad \frac{\frac{\frac{}{a : \alpha_2 \blacktriangleright a : \alpha_2; \emptyset} \quad \frac{}{\text{idle} \blacktriangleright \emptyset; \emptyset} \text{[I-IDLE]}}{\text{idle} \blacktriangleright x : \gamma; \{\text{un}(\gamma)\}} \text{[I-WEAK]}}{a?(x).\text{idle} \blacktriangleright a : \alpha_2; \{\alpha_2 \hat{=} [\gamma]^{1+\varrho_3, 2\varrho_4}, \text{un}(\gamma)\}} \text{[I-IN]}}{\frac{a!3 \mid a?(x).\text{idle} \blacktriangleright a : \alpha; \{\alpha \hat{=} \alpha_1 + \alpha_2, \alpha_1 \hat{=} [\text{int}]^{2\varrho_1, 1+\varrho_2}, \alpha_2 \hat{=} [\gamma]^{1+\varrho_3, 2\varrho_4}, \text{un}(\gamma)\}} \text{[I-PAR]}}{\text{new a in (a!3 | a?(x).idle) \blacktriangleright \emptyset; \{\alpha \hat{=} [\delta]^{2\varrho_1, 1+\varrho_2}, \alpha \hat{=} \alpha_1 + \alpha_2, \dots\}} \text{[I-NEW]}
\end{array}$$

The synthesized environment is empty, since the process has no free names, and the resulting constraint set is

$$\{\alpha \hat{=} [\delta]^{2\varrho_1, 1+\varrho_2}, \alpha \hat{=} \alpha_1 + \alpha_2, \alpha_1 \hat{=} [\text{int}]^{2\varrho_1, 1+\varrho_2}, \alpha_2 \hat{=} [\gamma]^{1+\varrho_3, 2\varrho_4}, \text{un}(\gamma)\}$$

Observe that a is used twice and each occurrence is assigned a distinct type variable α_i . Eventually, the reconstruction algorithm finds out that the *same* channel a is used simultaneously in different parts of the process, so it records the fact that the overall type α of a must be the combination of α_1 and α_2 in the constraint $\alpha \hat{=} \alpha_1 + \alpha_2$.

A solution for the obtained constraint set is the substitution

$$\{\alpha \mapsto [\text{int}]^{1,1}, \alpha_1 \mapsto [\text{int}]^{0,1}, \alpha_2 \mapsto [\text{int}]^{1,0}, \gamma \mapsto \text{int}, \delta \mapsto \text{int}, \varrho_{1..4} \mapsto 0, \varrho_5 \mapsto 1\}$$

confirming that a is a linear channel. This is not the only solution of the constraint set: another one can be obtained by setting all the use variables to ω , although in this case a is not recognized as a linear channel.

Note also that the application of [I-IN] is possible only if the name x of the received message occurs in the environment synthesized for the continuation process `idle`. Since the continuation process contains no occurrence of x , this name can only be introduced using [I-WEAK]. In general, [I-WEAK] is necessary to prove the completeness of the reconstruction algorithm as stated in Theorem 4.3. For example, $x : \mathbf{int} \vdash \mathbf{idle}$ is derivable according to the rules in Table 4, but as we have seen in the above derivation the reconstruction algorithm without [I-WEAK] would synthesize for `idle` an empty environment, not containing an association for x . \blacksquare

Example 4.5. We compute the constraint set of a simple process that accesses the same composite structure containing linear values. The process in Example 2.1 is too large to be discussed in full, so we consider the following, simpler process

$$\mathbf{fst}(x)?(y) . \mathbf{snd}(x)!(y+1)$$

which uses a pair x of channels and sends on the second channel in the pair the successor of the number received from the first channel (we assume that the language and the type reconstruction algorithm have been extended in the obvious way to support operations on numbers such as addition). We derive

$$\frac{\frac{}{x : \alpha_1 \blacktriangleright x : \alpha_1; \emptyset} \text{[I-NAME]}}{\mathbf{fst}(x) : \beta_1 \blacktriangleright x : \alpha_1; \{\alpha_1 \hat{=} \beta_1 \times \beta_2, \mathbf{un}(\beta_2)\}} \text{[I-FST]}}$$

for the first projection of x and

$$\frac{\frac{}{x : \alpha_2 \blacktriangleright x : \alpha_2; \emptyset} \text{[I-NAME]}}{\mathbf{snd}(x) : \gamma_2 \blacktriangleright x : \alpha_2; \{\alpha_2 \hat{=} \gamma_1 \times \gamma_2, \mathbf{un}(\gamma_1)\}} \text{[I-SND]}}$$

for the second projection of x . For the output operation we derive

$$\frac{\frac{\frac{}{y : \delta \blacktriangleright y : \delta; \emptyset} \text{[I-NAME]}}{\vdots} \quad \frac{}{1 : \mathbf{int} \blacktriangleright \emptyset; \emptyset} \text{[I-INT]}}{y+1 : \mathbf{int} \blacktriangleright y : \delta; \{\delta \hat{=} \mathbf{int}\}}}{\mathbf{snd}(x)!(y+1) \blacktriangleright x : \alpha_2, y : \delta; \{\alpha_2 \hat{=} \gamma_1 \times \gamma_2, \mathbf{un}(\gamma_1), \gamma_2 \hat{=} [\mathbf{int}]^{2e_3, 1+e_4}, \delta \hat{=} \mathbf{int}\}} \text{[I-OUT]}$$

so for the whole process we obtain

$$\frac{\frac{\frac{}{\vdots}}{\mathbf{fst}(x)?(y) . \mathbf{snd}(x)!(y+1) \blacktriangleright x : \alpha; \{\alpha \hat{=} \alpha_1 + \alpha_2, \alpha_1 \hat{=} \beta_1 \times \beta_2, \alpha_2 \hat{=} \gamma_1 \times \gamma_2, \beta_1 \hat{=} [\delta]^{1+e_1, 2e_2}, \gamma_2 \hat{=} [\mathbf{int}]^{2e_3, 1+e_4}, \mathbf{un}(\beta_2), \mathbf{un}(\gamma_1), \delta \hat{=} \mathbf{int}\}} \text{[I-IN]}}$$

Like in Example 4.4, here too the variable x is used multiple times and each occurrence is assigned a distinct type variable α_i , but this time such type variables must be assigned with a pair type in order for the constraint set to be solved. \blacksquare

5. CONSTRAINT SOLVING

In this section we describe an algorithm that determines whether a given constraint set \mathcal{C} is satisfiable and, if this is the case, computes a solution for \mathcal{C} . Among all possible solutions for \mathcal{C} , we strive to find one that allows us to identify as many linear channels as possible. To this aim, it is convenient to recall the notion of solution preciseness from [10].

Definition 5.1 (solution preciseness). Let \leq be the total order on uses such that $0 \leq 1 \leq \omega$. Given two solutions σ_1 and σ_2 for a constraint set \mathcal{C} , we say that σ_1 is *more precise* than σ_2 if $\sigma_1(\varrho) \leq \sigma_2(\varrho)$ for every $\varrho \in \text{expr}(\mathcal{C})$.

Roughly speaking, the preciseness of a solution is measured in terms of the numbers of unused and linear channels it identifies, which are related to the number of use variables assigned to 0 and 1. We will use Definition 5.1 as a guideline for developing our algorithm, although the algorithm may be unable to find *the* most precise solution. There are two reasons for this. First, there can be solutions with minimal use assignments that are incomparable according to Definition 5.1. This is related to the fact that the type system presented in Section 3 lacks the principal typing property. Second, to ensure termination when constraints concern infinite types, our algorithm makes some simplifying assumptions that may – in principle – imply a loss of precision of the resulting solution (see Example 5.11). Despite this, experience with the implementation suggests that the algorithm is indeed capable of identifying as many unused and linear channels as possible in practical situations, even when infinite types are involved. Before embarking in the technical description of the algorithm, we survey the key issues that we have to address and how they are addressed.

5.1. Overview. We begin by considering again the simple process below

$$\mathbf{new\ } a \mathbf{\ in\ } (a!3 \mid a?(x).\mathbf{idle}) \quad (5.1)$$

for which we have shown the reconstruction algorithm at work in Example 4.4. The process contains three occurrences of the channel a , two of them in subject position for input/output operations and one binding occurrence in the **new** construct. We have seen that the constraint generation algorithm associates the two rightmost occurrences of a with two type variables α_1 and α_2 that must respectively satisfy the constraints

$$\alpha_1 \hat{=} [\mathbf{int}]^{2\varrho_1, 1+\varrho_2} \quad (5.2)$$

$$\alpha_2 \hat{=} [\gamma]^{1+\varrho_3, 2\varrho_4} \quad (5.3)$$

whereas the leftmost occurrence of a has a type α which must satisfy the constraints

$$\alpha \hat{=} \alpha_1 + \alpha_2 \quad (5.4)$$

$$\alpha \hat{=} [\delta]^{\varrho_5, \varrho_5} \quad (5.5)$$

Even if none of these constraints concerns use variables directly, use variables are subject to implicit constraints that should be taken into account for finding a precise solution. To expose such implicit constraints, observe that in this first example we are in the fortunate situation where the type variables α , α_1 , and α_2 occur on the left-hand side of a constraint of the form $\beta \hat{=} \mathsf{T}$ where T is different from a type variable. In this case we say that β is *defined* and we call T its *definition*. If we substitute each type variable in (5.4) with its definition we obtain

$$[\delta]^{\varrho_5, \varrho_5} \hat{=} [\mathbf{int}]^{2\varrho_1, 1+\varrho_2} + [\gamma]^{1+\varrho_3, 2\varrho_4}$$

that reveals the relationships between the use variables. Knowing how type combination operates (Definition 3.5), we can derive two constraints concerning use variables

$$\begin{aligned}\varrho_5 &\hat{=} 2\varrho_1 + 1 + \varrho_3 \\ \varrho_5 &\hat{=} 1 + \varrho_2 + 2\varrho_4\end{aligned}$$

for which it is easy to figure out a solution that includes the substitutions $\{\varrho_{1..4} \mapsto 0, \varrho_5 \mapsto 1\}$ (see Example 4.4). No substitution can be more precise than this one hence such solution, which identifies a as a linear channel, is in fact optimal.

Let us now consider the following variation of (5.1)

$$a!3 \mid a?(x). \mathbf{idle}$$

where we have removed the restriction. In this case the generated constraints are the same (5.2), (5.3), and (5.4) as above, except that there is no constraint (5.5) that provides a definition for α . In a sense, α is defined because we know that it must be the combination of α_1 and α_2 for which we do have definitions. However, in order to come up with a general strategy for solving constraint sets, it is convenient to *complete* the constraint set with a defining equation for α : we know that α must be a channel type with messages of type **int**, because that is the shape of the definition for α_1 , but we do not know precisely the overall uses of α . Therefore, we generate a new constraint defining the *structure* of the type α , but with fresh use variables ϱ_5 and ϱ_6 in place of the unknown uses:

$$\alpha \hat{=} [\mathbf{int}]^{\varrho_5, \varrho_6}$$

We can now proceed as before, by substituting all type variables in (5.4) with their definition and deriving the use constraints below:

$$\begin{aligned}\varrho_5 &\hat{=} 2\varrho_1 + 1 + \varrho_3 \\ \varrho_6 &\hat{=} 1 + \varrho_2 + 2\varrho_4\end{aligned}$$

Note that, unlike in (5.1), we do not know whether ϱ_5 and ϱ_6 are required to be equal or not. Here we are typing an open process which, in principle, may be composed in parallel with other uses of the same channel a . Nonetheless, we can easily find a solution analogous to the previous one but with the use assignments $\{\varrho_{1..5} \mapsto 0, \varrho_{5,6} \mapsto 1\}$.

The idea of completing constraints with missing definitions is a fundamental ingredient of our constraint solving technique. In the previous example, completion was somehow superfluous because we could have obtained a definition for α by combining the definitions of α_1 and α_2 , which were available. However, completion allowed us to patch the constraint set so that it could be handled as in the previous case of process (5.1). In fact, it is easy to find processes for which completion becomes essential. Consider for example

$$\mathbf{new} \ a \ \mathbf{in} \ (a!3 \mid b!a) \tag{5.6}$$

where the bound channel a is used once for an output and then extruded through a free channel b . For this process, the reconstruction algorithm infers the type environment $b : \beta$ and the constraints below:

$$\begin{aligned}\alpha_1 &\hat{=} [\mathbf{int}]^{2\varrho_1, 1+\varrho_2} \\ \beta &\hat{=} [\alpha_2]^{2\varrho_3, 1+\varrho_4} \\ \alpha &\hat{=} \alpha_1 + \alpha_2 \\ \alpha &\hat{=} [\delta]^{\varrho_5, \varrho_6}\end{aligned}$$

where the three occurrences of a are associated from left to right with the type variables α , α_1 , and α_2 (Section C gives the derivation for (5.6)). Note that there is no constraint that defines α_2 . In fact, there is just no constraint with α_2 on the left hand side at all. The only hint that we have concerning α_2 is that it must yield α when combined with α_1 . Therefore, according to the definition of type combination, we can once more deduce that α_2 shares the same structure as α and α_1 and we can complete the set of constraints with

$$\alpha_2 \hat{=} [\text{int}]^{\varrho_6, \varrho_7}$$

where ϱ_6 and ϱ_7 are fresh use variables.

After performing the usual substitutions, we can finally derive the use constraints

$$\begin{aligned} \varrho_5 &\hat{=} 2\varrho_1 + \varrho_6 \\ \varrho_5 &\hat{=} 1 + \varrho_2 + \varrho_7 \end{aligned}$$

for which we find a solution including the assignments $\{\varrho_{1..4,7} \mapsto 0, \varrho_{5,6} \mapsto 1\}$. The interesting fact about this solution is the substitution $\varrho_6 \mapsto 1$, meaning that the constraint solver has inferred an input operation for the rightmost occurrence of a in (5.6), even though there is no explicit evidence of this operation in the process itself. The input operation is deduced “by subtraction”, seeing that a is used once in (5.6) for an output operation and knowing that a restricted (linear) channel like a must also be used for a matching input operation.

Note also that this is not the only possible solution for the use constraints. If, for example, it turns out that the extruded occurrence of a is never used (or is used twice) for an input, it is possible to obtain various solutions that include the assignments $\{\varrho_{5,6} \mapsto \omega\}$. However, the solution we have found above is the most precise according to Definition 5.1.

It is not always possible to find the most precise solution. This can be seen in the following variation of (5.6)

$$\text{new } a \text{ in } (a!3 \mid b!a \mid c!a) \tag{5.7}$$

where a is extruded twice, on b and on c (Section C gives the derivation). Here, as in (5.6), an input use for a is deduced “by subtraction”, but there is an ambiguity as to whether such input capability is transmitted through b or through c . Hence, there exist two incomparable solutions for the constraint set generated for (5.6). The lack of an optimal solution in general (hence of a principal typing) is a consequence of the condition imposing equal uses for restricted channels (see [T-NEW] and [I-NEW]). Without this condition, it would be possible to find the most precise solution for the constraints generated by (5.7) noticing that a is *never* explicitly used for an input operation, and therefore its input use could be 0. We think that this approach hinders the applicability of the reconstruction algorithm in practice, where separate compilation and type reconstruction of large programs are real concerns. We will elaborate more on this in Example 7.3. For the time being, let us analyze one last example showing a feature that we do *not* handle in our type system, namely polymorphism. The process

$$a?(x).b!x \tag{5.8}$$

models a forwarder that receives a message x from a and sends it on b . For this process the constraint generation algorithm yields the environment $a : \alpha, b : \beta$ and the constraints

$$\begin{aligned} \alpha &\hat{=} [\gamma]^{1+\varrho_1, 2\varrho_2} \\ \beta &\hat{=} [\gamma]^{2\varrho_3, 1+\varrho_4} \end{aligned}$$

(Section C gives the complete derivation). In particular, there is no constraint concerning the type variable γ and for good reasons: since the message x is only passed around in (5.8)

$\frac{[C\text{-AXIOM}]}{\mathcal{C} \cup \{\varphi\} \Vdash \varphi}$	$\frac{[C\text{-REFL}]}{\mathcal{C} \Vdash T \hat{R} T} \quad T \in \text{expr}(\mathcal{C})$	$\frac{[C\text{-SYMM}]}{\mathcal{C} \Vdash S \hat{R} T} \quad \mathcal{C} \Vdash T \hat{R} S$	$\frac{[C\text{-TRANS}]}{\mathcal{C} \Vdash T \hat{R} S} \quad \mathcal{C} \Vdash T \hat{R} T' \quad \mathcal{C} \Vdash T' \hat{R} S$
$\frac{[C\text{-COH 1}]}{\mathcal{C} \Vdash T \hat{R} S} \quad \mathcal{C} \Vdash T \hat{=} S$	$\frac{[C\text{-COH 2}]}{\mathcal{C} \Vdash T \hat{=} S_i} \quad \mathcal{C} \Vdash T \hat{=} S_1 + S_2 \quad i \in \{1, 2\}$	$\frac{[C\text{-CONG 1}]}{\mathcal{C} \Vdash T \hat{=} S} \quad \mathcal{C} \Vdash [T]^{U_1, U_2} \hat{\sim} [S]^{V_1, V_2}$	
$\frac{[C\text{-CONG 2}]}{\mathcal{C} \Vdash T_i \hat{R} S_i} \quad \mathcal{C} \Vdash T_1 \odot T_2 \hat{R} S_1 \odot S_2 \quad i \in \{1, 2\}$	$\frac{[C\text{-CONG 3}]}{\mathcal{C} \Vdash T_i \hat{=} S_i + S_{i+2}} \quad \mathcal{C} \Vdash T_1 \odot T_2 \hat{=} S_1 \odot S_2 + S_3 \odot S_4 \quad i \in \{1, 2\}$		
$\frac{[C\text{-SUBST}]}{\mathcal{C} \Vdash S_1 \hat{=} S_2 + S_3} \quad \mathcal{C} \Vdash T_1 \hat{=} T_2 + T_3 \quad \mathcal{C} \Vdash T_i \hat{=} S_i \quad (1 \leq i \leq 3)$			
$\frac{[C\text{-USE 1}]}{\mathcal{C} \Vdash U_i \hat{=} V_i} \quad \mathcal{C} \Vdash [T]^{U_1, U_2} \hat{=} [S]^{V_1, V_2} \quad i \in \{1, 2\}$	$\frac{[C\text{-USE 2}]}{\mathcal{C} \Vdash U_i \hat{=} V_i + V_{i+2}} \quad \mathcal{C} \Vdash [T]^{U_1, U_2} \hat{=} [S_1]^{V_1, V_2} + [S_2]^{V_3, V_4} \quad i \in \{1, 2\}$		

Table 8: Constraint deduction system.

but never actually used, the channels a and b should be considered *polymorphic*. Note that in this case we know nothing about the structure of γ hence completion of the constraint set is not applicable. In this work we do not deal with polymorphism and will refrain from solving sets of constraints where there is no (structural) information for unconstrained type variables. Just observe that handling polymorphism is not simply a matter of allowing (universally quantified) type variables in types. For example, a type variable involved in a constraint $\alpha \hat{=} \alpha + \alpha$ does not have any structural information and therefore is polymorphic, but can only be instantiated with unlimited types. The implementation has a defaulting mechanism that forces unconstrained type variables to a base type.

We now formalize the ideas presented so far into an algorithm, for which we have already identified the key phases: the ability to recognize types that “share the same structure”, which we call *structurally coherent* (Definition 3.6); the *completion* of a set of constraints with “missing definitions” so that each type variable has a proper definition; the derivation and solution of *use constraints*. Let us proceed in order.

5.2. Verification. In Section 5.1 we have seen that some constraints can be *derived* from the ones produced during the constraint generation phase (Section 4). We now define a deduction system that, starting from a given constraint set \mathcal{C} , computes all the “derivable facts” about the types in $\text{expr}(\mathcal{C})$. Such deduction system is presented as a set of inference rules in Table 8, where \mathcal{R} ranges over the symbols $=$ and \sim . Each rule derives a judgment of the form $\mathcal{C} \Vdash \varphi$ meaning that the constraint φ is derivable from those in \mathcal{C} (Proposition 5.2 below formalizes this property). Rule [C-AXIOM] simply takes each constraint in \mathcal{C} as an axiom. Rules [C-REFL], [C-SYMM], and [C-TRANS] state the obvious reflexivity, symmetry, and

transitivity of $=$ and \sim . Rules [C-COH 1] and [C-COH 2] deduce coherence relations: equality implies coherence, for $= \sqsubseteq \sim$, and each component of a combination is coherent to the combination itself (and therefore, by transitivity, to the other component). Rules [C-CONG 1] through [C-CONG 3] state congruence properties of $=$ and \sim which follow directly from Definition 3.5: when two channel types are coherent, their message types must be equal; corresponding components of \mathcal{R} -related composite types are \mathcal{R} -related. Rule [C-SUBST] allows the substitution of equal types in combinations. Finally, [C-USE 1] and [C-USE 2] allow us to deduce *use constraints* of the form $U \hat{=} V$ involving use variables. Both rules are self-explanatory and follow directly from Definition 3.5.

We state two important properties of this deduction system:

Proposition 5.2. *Let $\mathcal{C} \Vdash \varphi$. The following properties hold:*

- (1) \mathcal{C} and $\mathcal{C} \cup \{\varphi\}$ are equivalent (see Definition 4.1).
- (2) $\text{expr}(\mathcal{C}) = \text{expr}(\mathcal{C} \cup \{\varphi\})$.

Proof. A simple induction on the derivation of $\mathcal{C} \Vdash \varphi$. □

The first property confirms that all the derivable relations are already encoded in the original constraint set, in a possibly implicit form. The deduction system makes them explicit. The second property assures us that no new type expressions are introduced by the deduction system. Since the inference rules in Section 4 always generate finite constraint sets, this implies that the set of all derivable constraints is also finite and can be computed in finite time. This is important because the presence or absence of particular constraints determines the (un)satisfiability of a constraint set:

Proposition 5.3. *If $\mathcal{C} \Vdash T \sim S$ where T and S are proper type expressions with different topmost constructors, then \mathcal{C} has no solution.*

Proof. Suppose by contradiction that σ is a solution for \mathcal{C} . By Proposition 5.2(1) we have that σ is also a solution for $\mathcal{C} \cup \{T \sim S\}$. This is absurd, for if T and S have different topmost constructors, then so do σT and σS , hence $\sigma T \not\sim \sigma S$. □

The converse of Proposition 5.3 is not true in general. For example, the constraint set $\{[\text{int}]^{0,1} \hat{=} [\text{int}]^{1,0}\}$ has no solution because of the implicit constraints on corresponding uses and yet it satisfies the premises of Proposition 5.3. However, when \mathcal{C} is a constraint set generated by the inference rules in Section 4, the converse of Proposition 5.3 holds. This means that we can use structural coherence as a necessary and sufficient condition for establishing the satisfiability of constraint sets generated by the reconstruction algorithm.

Before proving this fact we introduce some useful notation. For $\mathcal{R} \in \{=, \sim\}$ let

$$\mathcal{R}_{\mathcal{C}} \stackrel{\text{def}}{=} \{(T, S) \mid \mathcal{C} \Vdash T \hat{\mathcal{R}} S\}$$

and observe that $\mathcal{R}_{\mathcal{C}}$ is an equivalence relation on $\text{expr}(\mathcal{C})$ by construction, because of the rules [C-REFL], [C-SYMM], and [C-TRANS]. Therefore, it partitions the type expressions in \mathcal{C} into \mathcal{R} -equivalence classes. Now, we need some way to *choose*, from each \mathcal{R} -equivalence class, *one* representative element of the class. To this aim, we fix a total order \sqsubseteq between type expressions such that $T \sqsubseteq \alpha$ for every proper T and every α and we define:¹

¹In a Haskell or OCaml implementation such total order could be, for instance, the one automatically defined for the algebraic data type that represents type expressions and where the value constructor representing type variables is the last one in the data type definition.

Definition 5.4 (canonical representative). Let $\text{crep}_{\mathcal{R}}(\mathcal{C}, \mathbb{T})$ be the \sqsubseteq -least type expression \mathbb{S} such that $\mathbb{T} \mathcal{R}_{\mathcal{C}} \mathbb{S}$. We say that $\text{crep}_{\mathcal{R}}(\mathcal{C}, \mathbb{T})$ is the *canonical representative* of \mathbb{T} with respect to the relation $\mathcal{R}_{\mathcal{C}}$.

Note that, depending on \sqsubseteq , we may have different definitions of $\text{crep}_{\mathcal{R}}(\mathcal{C}, \mathbb{T})$. The exact choice of the canonical representative does not affect the ability of the algorithm to compute a solution for a constraint set (Theorem 5.12) although – in principle – it may affect the precision of the solution (Example 5.11). Note also that, because of the assumption we have made on the total order \sqsubseteq , $\text{crep}_{\mathcal{R}}(\mathcal{C}, \mathbb{T})$ is proper whenever \mathbb{T} is proper or when \mathbb{T} is some type variable α such that there is a “definition” for α in \mathcal{C} . In fact, it is now time to define precisely the notion of *defined* and *undefined* type variables:

Definition 5.5 (defined and undefined type variables). Let

$$\begin{aligned} \text{def}_{\mathcal{R}}(\mathcal{C}) &\stackrel{\text{def}}{=} \{ \alpha \in \text{expr}(\mathcal{C}) \mid \text{crep}_{\mathcal{R}}(\mathcal{C}, \alpha) \text{ is proper} \} \\ \text{undef}_{\mathcal{R}}(\mathcal{C}) &\stackrel{\text{def}}{=} \{ \alpha \in \text{expr}(\mathcal{C}) \setminus \text{def}_{\mathcal{R}}(\mathcal{C}) \} \end{aligned}$$

We say that α is \mathcal{R} -defined or \mathcal{R} -undefined in \mathcal{C} according to $\alpha \in \text{def}_{\mathcal{R}}(\mathcal{C})$ or $\alpha \in \text{undef}_{\mathcal{R}}(\mathcal{C})$.

We can now prove that the coherence check is also a sufficient condition for satisfiability.

Proposition 5.6. *Let $P \blacktriangleright \Delta; \mathcal{C}$. If $\mathcal{C} \Vdash \mathbb{T} \sim \mathbb{S}$ where \mathbb{T} and \mathbb{S} are proper type expressions implies that \mathbb{T} and \mathbb{S} have the same topmost constructor, then \mathcal{C} has a solution.*

Proof. We only sketch the proof, since we will prove a more general result later on (see Theorem 5.12). Consider the use substitution $\sigma_{use} \stackrel{\text{def}}{=} \{ \varrho \mapsto \omega \mid \varrho \in \text{expr}(\mathcal{C}) \}$ mapping all use variables in \mathcal{C} to ω , let Σ be the system of equations $\{ \alpha_i = \mathbb{T}_i \mid 1 \leq i \leq n \}$ defined by

$$\Sigma \stackrel{\text{def}}{=} \{ \alpha = \sigma_{use} \text{crep}_{\sim}(\mathcal{C}, \alpha) \mid \alpha \in \text{def}_{\sim}(\mathcal{C}) \} \cup \{ \alpha = \mathbf{int} \mid \alpha \in \text{undef}_{\sim}(\mathcal{C}) \}$$

and observe that every \mathbb{T}_i is a proper type expression. From Theorem 3.2 we know that Σ has a unique solution $\sigma_{type} = \{ \alpha_i \mapsto t_i \mid 1 \leq i \leq n \}$ such that $t_i = \sigma_{type} \mathbb{T}_i$ for every $1 \leq i \leq n$. It only remains to show that $\sigma_{use} \cup \sigma_{type}$ is a solution for \mathcal{C} . This follows from two facts: (1) from the hypothesis $P \blacktriangleright \Delta; \mathcal{C}$ we know that all channel types in \mathcal{C} have one use variable in each of their use slots, hence the substitution σ_{use} forces all uses to ω ; (2) from the hypothesis and the rules [C-CONG *] we know that all proper type expressions in the same (\sim) -equivalence class have the same topmost constructor. \square

In Proposition 5.6, for finding a substitution for all the type variables in \mathcal{C} , we default each type variable in $\text{undef}_{\sim}(\mathcal{C})$ to \mathbf{int} . This substitution is necessary in order to satisfy the constraints $\text{un}(\alpha)$, namely those of the form $\alpha \hat{=} \alpha + \alpha$, when $\alpha \in \text{undef}_{\sim}(\mathcal{C})$. These α ’s are the “polymorphic type variables” that we have already discussed earlier. Since we leave polymorphism for future work, in the rest of this section we make the assumption that $\text{undef}_{\sim}(\mathcal{C}) = \emptyset$, namely that all type variables are (\sim) -defined.

Example 5.7. Below is a summary of the constraint set \mathcal{C} generated in Example 4.5:

$$\begin{array}{llll} \alpha \hat{=} \alpha_1 + \alpha_2 & \alpha_1 \hat{=} \beta_1 \times \beta_2 & \beta_1 \hat{=} [\delta]^{1+e_1, 2e_2} & \gamma_1 \hat{=} \gamma_1 + \gamma_1 \\ \delta \hat{=} \mathbf{int} & \alpha_2 \hat{=} \gamma_1 \times \gamma_2 & \beta_2 \hat{=} \beta_2 + \beta_2 & \gamma_2 \hat{=} [\mathbf{int}]^{2e_3, 1+e_4} \end{array}$$

Note that $\{\alpha, \beta_2, \gamma_1\} \subseteq \text{def}_{\sim}(\mathcal{C}) \setminus \text{def}_{=}(\mathcal{C})$. In particular, they all have a proper canonical representative, which we may assume to be the following ones:

$$\begin{aligned} \text{crep}_{\sim}(\mathcal{C}, \alpha) &= \text{crep}_{\sim}(\mathcal{C}, \alpha_1) = \text{crep}_{\sim}(\mathcal{C}, \alpha_2) = \beta_1 \times \beta_2 \\ \text{crep}_{\sim}(\mathcal{C}, \beta_1) &= \text{crep}_{\sim}(\mathcal{C}, \gamma_1) = [\delta]^{1+\varrho_1, 2\varrho_2} \\ \text{crep}_{\sim}(\mathcal{C}, \beta_2) &= \text{crep}_{\sim}(\mathcal{C}, \gamma_2) = [\mathbf{int}]^{2\varrho_3, 1+\varrho_4} \\ \text{crep}_{\sim}(\mathcal{C}, \delta) &= \mathbf{int} \end{aligned}$$

It is immediate to verify that the condition of Proposition 5.6 holds, hence we conclude that \mathcal{C} is satisfiable. Indeed, a solution for \mathcal{C} is

$$\{\alpha, \alpha_{1,2} \mapsto [\mathbf{int}]^{\omega, \omega} \times [\mathbf{int}]^{\omega, \omega}, \beta_{1,2}, \gamma_{1,2} \mapsto [\mathbf{int}]^{\omega, \omega}, \delta \mapsto \mathbf{int}, \varrho_{1..4} \mapsto \omega\}$$

even though we will find a more precise solution in Example 5.13. \blacksquare

5.3. Constraint set completion. If the satisfiability of the constraint set is established (Proposition 5.6), the subsequent step is its completion in such a way that every type variable α has a definition in the form of a constraint $\alpha \hat{=} \mathbb{T}$ where \mathbb{T} is proper. Recall that this step is instrumental for discovering all the (implicit) use constraints.

In Example 5.7 we have seen that some type variables may be (\sim)-defined but ($=$)-undefined. The \sim relation provides information about the structure of the type that should be assigned to the type variable, but says nothing about the uses in them. Hence, the main task of completion is the creation of fresh use variables for those channel types of which only the structure is known. In the process, fresh type variables need to be created as well, and we should make sure that all such type variables are ($=$)-defined to guarantee that completion eventually terminates. We will be able to do this, possibly at the cost of some precision of the resulting solution.

We begin the formalization of completion by introducing an injective function \mathbf{t} that, given a pair of type variables α and β , creates a new type variable $\mathbf{t}(\alpha, \beta)$. We assume that $\mathbf{t}(\alpha, \beta)$ is different from any type variable generated by the algorithm in Section 4 so that the type variables obtained through \mathbf{t} are effectively fresh. Then we define an *instantiation* function instance that, given a type variable α and a type expression \mathbb{T} , produces a new type expression that is structurally coherent to \mathbb{T} , but where all use expressions and type variables have been respectively replaced by fresh use and type variables. The first argument α of instance records the fact that such instantiation is necessary for completing α . Formally:

$$\text{instance}(\alpha, \mathbb{T}) \stackrel{\text{def}}{=} \begin{cases} \mathbf{t}(\alpha, \beta) & \text{if } \mathbb{T} = \beta \\ \mathbf{int} & \text{if } \mathbb{T} = \mathbf{int} \\ [\mathbf{S}]^{\varrho_1, \varrho_2} & \text{if } \mathbb{T} = [\mathbf{S}]^{\mathbf{U}, \mathbf{V}}, \varrho_i \text{ fresh} \\ \text{instance}(\alpha, \mathbb{T}_1) \odot \text{instance}(\alpha, \mathbb{T}_2) & \text{if } \mathbb{T} = \mathbb{T}_1 \odot \mathbb{T}_2 \end{cases} \quad (5.9)$$

All the equations but the first one are easily explained: the instance of \mathbf{int} cannot be anything but \mathbf{int} itself; the instance of a channel type $[\mathbf{S}]^{\mathbf{U}, \mathbf{V}}$ is the type expression $[\mathbf{S}]^{\varrho_1, \varrho_2}$ where we generate two fresh use variables corresponding to \mathbf{U} and \mathbf{V} ; the instance of a composite type $\mathbb{T} \odot \mathbb{S}$ is the composition of the instances of \mathbb{T} and \mathbb{S} . For example, we have

$$\text{instance}(\alpha, \beta \times [[\mathbf{int}]^{\mathbf{U}_1, \mathbf{U}_2}]^{\mathbf{V}_1, \mathbf{V}_2}) = \mathbf{t}(\alpha, \beta) \times [[\mathbf{int}]^{\mathbf{U}_1, \mathbf{U}_2}]^{\varrho_1, \varrho_2}$$

where ϱ_1 and ϱ_2 are fresh. Note that, while instantiating a channel type $[S]^{U,V}$, there is no need to instantiate S because $[t]^{\kappa_1, \kappa_2} \sim [s]^{\kappa_3, \kappa_4}$ implies $t = s$ so S is *exactly* the message type we must use in the instance of $[S]^{U,V}$.

Concerning the first equation in (5.9), in principle we want $\text{instance}(\alpha, \beta)$ to be the same as $\text{instance}(\alpha, \text{crep}_{\sim}(\mathcal{C}, \beta))$, but doing so directly would lead to an ill-founded definition for instance , since nothing prevents β from occurring in $\text{crep}_{\sim}(\mathcal{C}, \beta)$ (types can be infinite). We therefore instantiate β to a new type variable $\mathfrak{t}(\alpha, \beta)$ which will in turn be defined by a new constraint $\mathfrak{t}(\alpha, \beta) \hat{=} \text{instance}(\alpha, \text{crep}_{\sim}(\mathcal{C}, \beta))$.

There are a couple of subtleties concerning the definition of instance . The first one is that, strictly speaking, instance is a relation rather than a function because the fresh use variables in (5.9) are not uniquely determined. In practice, instance can be turned into a proper function by devising a deterministic mechanism that picks fresh use variables in a way similar to the \mathfrak{t} function that we have defined above. The formal details are tedious but well understood, so we consider the definition of instance above satisfactory as is. The second subtlety is way more serious and has to do with the instantiation of type variables (first equation in (5.9)) which hides a potential approximation due to this completion phase. To illustrate the issue, suppose that

$$\alpha \sim [\text{int}]^{U,V} \times \alpha \quad (5.10)$$

is the only constraint concerning α in some constraint set \mathcal{C} so that we need to provide a (=)-definition for α . According to (5.9) we have

$$\text{instance}(\alpha, [\text{int}]^{U,V} \times \alpha) = [\text{int}]^{\varrho_1, \varrho_2} \times \mathfrak{t}(\alpha, \alpha)$$

so by adding the constraints

$$\alpha \hat{=} \mathfrak{t}(\alpha, \alpha) \quad \text{and} \quad \mathfrak{t}(\alpha, \alpha) \hat{=} [\text{int}]^{\varrho_1, \varrho_2} \times \mathfrak{t}(\alpha, \alpha) \quad (5.11)$$

we complete the definition for α . There is a fundamental difference between the constraint (5.10) and those in (5.11) in that the former admits far more solutions than those admitted by (5.11). For example, the constraint (5.10) can be satisfied by a solution that contains the assignment $\alpha \mapsto t$ where $t = [\text{int}]^{1,0} \times [\text{int}]^{0,1} \times t$, but the constraint (5.11) cannot. The problem of a constraint like (5.10) is that, when we only have structural information about a type variable, we have no clue about the uses in its definition, if they follow a pattern, and what the pattern is. In principle, in order to account for all the possibilities, we should generate fresh use variables in place of any use slot in the possibly infinite type. In practice, however, we want completion to eventually terminate, and the definition of instance given by (5.9) is one easy way to ensure this: what we are saying there is that each type variable β that contributes to the definition of a (=)-undefined type variable α is instantiated only once. This trivially guarantees completion termination, for there is only a finite number of distinct variables to be instantiated. The price we pay with this definition of instance is a potential loss of precision in the solution of use constraints. We say “potential” because we have been unable to identify a concrete example that exhibits such loss of precision. Part of the difficulty of this exercise is due to the fact that the effects of the approximation on the solution of use constraints may depend on the particular choice of canonical representatives, which is an implementation detail of the constraint solver (see Definition 5.4). In part, the effects of the approximation are limited to peculiar situations:

- (1) There is only a fraction of constraint sets where the same type variable occurring in several different positions must be instantiated, namely those having as solution types

with infinite branches containing only finitely many channel type constructors. The constraint (5.10) is one such example. In all the other cases, the given definition of instance does not involve any approximation.

- (2) A significant fraction of the type variables for which only structural information is known are those generated by the rules [I-FST], [I-SND], and [I-WEAK]. These type variables stand for *unlimited* types, namely for types whose uses are either 0 or ω . In fact, in most cases *all* the uses in these unlimited types are 0. Therefore, the fact that only a handful of fresh use variables is created, instead of infinitely many, does not cause any approximation at all, since the use variables in these type expressions would all be instantiated to 0 anyway.

We define the completion of a constraint set \mathcal{C} as the least superset of \mathcal{C} where all the (=)-undefined type variables in \mathcal{C} have been properly instantiated:

Definition 5.8 (completion). The *completion* of \mathcal{C} , written $\overline{\mathcal{C}}$, is the least set such that:

- (1) $\mathcal{C} \subseteq \overline{\mathcal{C}}$;
- (2) $\alpha \in \text{undef}_=(\mathcal{C})$ implies $\alpha \hat{=} \mathbf{t}(\alpha, \alpha) \in \overline{\mathcal{C}}$;
- (3) $\mathbf{t}(\alpha, \beta) \in \text{expr}(\overline{\mathcal{C}})$ implies $\mathbf{t}(\alpha, \beta) \hat{=} \text{instance}(\alpha, \text{crep}_{\sim}(\mathcal{C}, \beta)) \in \overline{\mathcal{C}}$.

The completion $\overline{\mathcal{C}}$ of a finite constraint set \mathcal{C} can always be computed in finite time as the number of necessary instantiations is bound by the square of the cardinality of $\text{undef}_=(\mathcal{C})$. Because of the approximation of instances for undefined variables, \mathcal{C} and $\overline{\mathcal{C}}$ are not equivalent in general (see Example 5.11 below). However, the introduction of instances does not affect the satisfiability of the set of constraints.

Proposition 5.9. *The following properties hold:*

- (1) *If \mathcal{C} is satisfiable, then $\overline{\mathcal{C}}$ is satisfiable.*
- (2) *If σ is a solution for $\overline{\mathcal{C}}$, then σ is also a solution for \mathcal{C} .*

Proof. Each (\sim)-equivalence class in $\overline{\mathcal{C}}$ contains exactly one (\sim)-equivalence class in \mathcal{C} , for each new type expression that has been introduced in $\overline{\mathcal{C}}$ is structurally coherent to an existing type expression in \mathcal{C} . Then item (1) is a consequence of Proposition 5.6, while item (2) follows from the fact that $\mathcal{C} \subseteq \overline{\mathcal{C}}$. \square

Example 5.10. Considering the constraint set \mathcal{C} in Example 5.7, we have three type variables requiring instantiation, namely α , β_2 , and γ_1 . According to Definition 5.8, and using the same canonical representatives mentioned in Example 5.7, we augment the constraint set with the constraints

$$\begin{array}{lll}
\alpha \hat{=} \mathbf{t}(\alpha, \alpha) & \mathbf{t}(\alpha, \alpha) \hat{=} \text{instance}(\alpha, \beta_1 \times \beta_2) & = \mathbf{t}(\alpha, \beta_1) \times \mathbf{t}(\alpha, \beta_2) \\
& \mathbf{t}(\alpha, \beta_1) \hat{=} \text{instance}(\alpha, [\delta]^{1+\varrho_1, 2\varrho_2}) & = [\delta]^{\varrho_5, \varrho_6} \\
& \mathbf{t}(\alpha, \beta_2) \hat{=} \text{instance}(\alpha, [\mathbf{int}]^{2\varrho_3, 1+\varrho_4}) & = [\mathbf{int}]^{\varrho_7, \varrho_8} \\
\beta_2 \hat{=} \mathbf{t}(\beta_2, \beta_2) & \mathbf{t}(\beta_2, \beta_2) \hat{=} \text{instance}(\beta_2, [\mathbf{int}]^{2\varrho_3, 1+\varrho_4}) & = [\mathbf{int}]^{\varrho_9, \varrho_{10}} \\
\gamma_1 \hat{=} \mathbf{t}(\gamma_1, \gamma_1) & \mathbf{t}(\gamma_1, \gamma_1) \hat{=} \text{instance}(\gamma_1, [\delta]^{1+\varrho_1, 2\varrho_2}) & = [\delta]^{\varrho_{11}, \varrho_{12}}
\end{array}$$

where the ϱ_i with $i \geq 5$ are all fresh.

Observe that the canonical (\sim)-representative of β_2 is instantiated twice, once for defining α and once for defining β_2 itself. We will see in Example 5.13 that this double instantiation is key for inferring that $\mathbf{snd}(x)$ in Example 4.5 is used linearly. \blacksquare

Example 5.11. In this example we show the potential effects of instantiation on the ability of the type reconstruction algorithm to identify linear channels. To this aim, consider the following constraint set

$$\begin{aligned}\alpha &\approx [\mathbf{int}]^{\mathbf{U},\mathbf{V}} \times \alpha \\ \beta &\hat{=} [\mathbf{int}]^{0,1+\varrho_1} \times [\mathbf{int}]^{0,2\varrho_2} \times \beta \\ \gamma &\hat{=} [\mathbf{int}]^{0,0} \times [\mathbf{int}]^{0,0} \times \gamma \\ \alpha &\hat{=} \beta + \gamma\end{aligned}$$

where, to limit the number of use variables without defeating the purpose of the example, we write the constant use 0 in a few use slots. Observe that this constraint set admits the solution $\{\alpha \mapsto t, \beta \mapsto t, \gamma \mapsto s, \varrho_{1,2} \mapsto 0\}$ where t and s are the types that satisfy the equalities $t = [\mathbf{int}]^{0,1} \times [\mathbf{int}]^{0,0} \times t$ and $s = [\mathbf{int}]^{0,0} \times s$. Yet, if we instantiate α following the procedure outlined above we obtain the constraints

$$\alpha \hat{=} \mathbf{t}(\alpha, \alpha) \quad \text{and} \quad \mathbf{t}(\alpha, \alpha) \hat{=} [\mathbf{int}]^{\varrho_3, \varrho_4} \times \mathbf{t}(\alpha, \alpha)$$

and now the two constraints below follow by the congruence rule [C-CONG *]:

$$\begin{aligned}[\mathbf{int}]^{\varrho_3, \varrho_4} &\hat{=} [\mathbf{int}]^{0,1+\varrho_1} + [\mathbf{int}]^{0,0} \\ [\mathbf{int}]^{\varrho_3, \varrho_4} &\hat{=} [\mathbf{int}]^{0,2\varrho_2} + [\mathbf{int}]^{0,0}\end{aligned}$$

This implies that the use variable ϱ_4 must simultaneously satisfy the constraints

$$\varrho_4 \hat{=} 1 + \varrho_1 \quad \text{and} \quad \varrho_4 \hat{=} 2\varrho_2$$

which is only possible if we assign ϱ_1 and ϱ_2 to a use other than 0 and ϱ_4 to ω . In other words, after completion the only feasible solutions for the constraint set above have the form $\{\alpha \mapsto t', \beta \mapsto t', \gamma \mapsto s, \varrho_{1,2} \mapsto \kappa, \varrho_3 \mapsto 0, \varrho_4 \mapsto \omega\}$ for $1 \leq \kappa$ where $t' = [\mathbf{int}]^{0,\omega} \times t'$, which are less precise than the one that we could figure out before the instantiation: t denotes an infinite tuple of channels in which those in odd-indexed positions are used for performing exactly one output operation; t' denotes an infinite tuple of channels, each being used for an unspecified number of output operations. \blacksquare

5.4. Solution synthesis. In this phase, substitutions are found for all the use and type variables that occur in a (completed) constraint set. We have already seen that it is always possible to consider a trivial use substitution that assigns each use variable to ω . In this phase, however, we have all the information for finding a use substitution that, albeit not necessarily optimal because of the approximation during the completion phase, is minimal according to the \leq precision order on uses of Definition 5.1.

The first step for computing a use substitution is to collect the whole set of constraints concerning use expressions. This is done by repeatedly applying the rules [C-USE 1] and [C-USE 2] shown in Table 8. Note that the set of derivable use constraints is finite and can be computed in finite time because \mathcal{C} is finite. Also, we are sure to derive *all* possible use constraints if we apply these two rules to a completed constraint set.

Once use constraints have been determined, any particular substitution for use variables can be found by means of an exhaustive search over all the possible substitutions: the number of such substitutions is finite because the number of use variables is finite and so is the domain $\{0, 1, \omega\}$ on which they range. Clearly this brute force approach is not practical in general and in Section 6 we will discuss two techniques that reduce the search space

for use substitutions. The main result of this section is independent of the particular use substitution σ_{use} that has been identified.

Theorem 5.12 (correctness of the constraint solving algorithm). *Let $P \blacktriangleright \Delta; \mathcal{C}$. If*

- (1) $\mathcal{C} \Vdash \mathbb{T} \sim \mathbb{S}$ where \mathbb{T} and \mathbb{S} are proper type expressions implies that \mathbb{T} and \mathbb{S} have the same topmost constructor, and
 - (2) σ_{use} is a solution of the use constraints of $\overline{\mathcal{C}}$, and
 - (3) σ_{type} is the solution of the system $\Sigma \stackrel{\text{def}}{=} \{\alpha = \sigma_{use} \text{crep}_=(\overline{\mathcal{C}}, \alpha) \mid \alpha \in \text{expr}(\overline{\mathcal{C}})\}$,
- then $\sigma_{use} \cup \sigma_{type}$ is a solution for \mathcal{C} .

Proof. Let $\sigma \stackrel{\text{def}}{=} \sigma_{use} \cup \sigma_{type}$. We have to prove the implications of Definition 4.1 for $\overline{\mathcal{C}}$. We focus on constraints of the form $\mathbb{T} \hat{=} \mathbb{S}_1 + \mathbb{S}_2$, the other constraints being simpler and/or handled in a similar way.

Let $\mathcal{R} \stackrel{\text{def}}{=} \{((\sigma\mathbb{S}_1, \sigma\mathbb{S}_2), \sigma\mathbb{T}) \mid \overline{\mathcal{C}} \Vdash \mathbb{T} \hat{=} \mathbb{S}_1 + \mathbb{S}_2\}$. It is enough to show that \mathcal{R} satisfies the conditions of Definition 3.5, since type combination is the *largest* relation that satisfies those same conditions. Suppose $((s_1, s_2), t) \in \mathcal{R}$. Then there exist \mathbb{T} , \mathbb{S}_1 , and \mathbb{S}_2 such that $\overline{\mathcal{C}} \Vdash \mathbb{T} \hat{=} \mathbb{S}_1 + \mathbb{S}_2$ and $t = \sigma\mathbb{T}$ and $s_i = \sigma\mathbb{S}_i$ for $i = 1, 2$. Without loss of generality, we may also assume that \mathbb{T} , \mathbb{S}_1 , and \mathbb{S}_2 are proper type expressions. Indeed, suppose that this is not the case and, for instance, $\mathbb{T} = \alpha$. Then, from [C-SUBST] we have that $\overline{\mathcal{C}} \Vdash \text{crep}_=(\overline{\mathcal{C}}, \alpha) \hat{=} \mathbb{S}_1 + \mathbb{S}_2$ and, since σ is a solution of Σ , we know that $\sigma(\alpha) = \sigma \text{crep}_=(\overline{\mathcal{C}}, \alpha)$. Therefore, the same pair $((s_1, s_2), t) \in \mathcal{R}$ can also be obtained from the triple $(\text{crep}_=(\overline{\mathcal{C}}, \alpha), \mathbb{S}_1, \mathbb{S}_2)$ whose first component is proper. The same argument applies for \mathbb{S}_1 and \mathbb{S}_2 .

Now we reason by cases on the structure of \mathbb{T} , \mathbb{S}_1 , and \mathbb{S}_2 , knowing that all these type expressions have the same topmost constructor from hypothesis (1) and [C-COH 2]:

- If $\mathbb{T} = \mathbb{S}_1 = \mathbb{S}_2 = \text{int}$, then condition (1) of Definition 3.5 is satisfied.
- If $\mathbb{T} = [\mathbb{T}']^{\mathbb{U}_1, \mathbb{U}_2}$ and $\mathbb{S}_i = [\mathbb{S}'_i]^{\mathbb{V}_{2i-1}, \mathbb{V}_{2i}}$ for $i = 1, 2$, then from [C-COH 2] and [C-CONG 1] we deduce $\overline{\mathcal{C}} \Vdash \mathbb{T}' \hat{=} \mathbb{S}'_i$ and from [C-USE 2] we deduce $\overline{\mathcal{C}} \Vdash \mathbb{U}_i \hat{=} \mathbb{V}_i + \mathbb{V}_{i+2}$ for $i = 1, 2$. Since σ is a solution for the equality constraints in $\overline{\mathcal{C}}$, we deduce $\sigma\mathbb{T}' = \sigma\mathbb{S}_1 = \sigma\mathbb{S}_2$. Since σ is a solution for the use constraints in $\overline{\mathcal{C}}$, we conclude $\sigma\mathbb{U}_i = \sigma\mathbb{V}_i + \sigma\mathbb{V}_{i+2}$ for $i = 1, 2$. Hence, condition (2) of Definition 3.5 is satisfied.
- If $\mathbb{T} = \mathbb{T}_1 \odot \mathbb{T}_2$ and $\mathbb{S}_i = \mathbb{S}_{i1} \odot \mathbb{S}_{i2}$, then from [C-CONG 3] we deduce $\overline{\mathcal{C}} \Vdash \mathbb{T}_i \hat{=} \mathbb{S}_{i1} + \mathbb{S}_{i2}$ for $i = 1, 2$. We conclude $((\sigma\mathbb{S}_{i1}, \sigma\mathbb{S}_{i2}), \sigma\mathbb{T}_i) \in \mathcal{R}$ by definition of \mathcal{R} , hence condition (3) of Definition 3.5 is satisfied. \square

Note that the statement of Theorem 5.12 embeds the constraint solving algorithm, which includes a verification phase (item (1)), a constraint completion phase along with an (unspecified, but effective) computation of a solution for the use constraints (item (2)), and the computation of a solution for the original constraint set in the form of a finite system of equations (item (3)). The conclusion of the theorem states that the algorithm is correct.

Example 5.13. There are three combination constraints in the set $\overline{\mathcal{C}}$ obtained in Example 5.10, namely $\alpha \hat{=} \alpha_1 + \alpha_2$, $\beta_2 \hat{=} \beta_2 + \beta_2$, and $\gamma_1 \hat{=} \gamma_1 + \gamma_1$. By performing suitable substitutions with [C-SUBST] we obtain

$$\frac{\overline{\mathcal{C}} \Vdash \alpha \hat{=} \alpha_1 + \alpha_2 \quad [\text{C-AXIOM}]}{\overline{\mathcal{C}} \Vdash \mathbf{t}(\alpha, \beta_1) \times \mathbf{t}(\alpha, \beta_2) \hat{=} \beta_1 \times \beta_2 + \gamma_1 \times \gamma_2 \quad [\text{C-SUBST}] \text{ (multiple applications)}} \quad [\text{C-CONG 3}]$$

$$\overline{\mathcal{C}} \Vdash \mathbf{t}(\alpha, \beta_i) \hat{=} \beta_i + \gamma_i$$

constraint set has been completed, substitutions in constraints expressing combinations can be performed efficiently by mapping each type variable to its canonical representative.

6.2. Use constraints resolution. In Section 5 we have refrained from providing any detail about how use constraints are solved and argued that a particular use substitution can always be found given that both the set of constraints and the domain of use variables are finite. While this argument suffices for establishing the decidability of this crucial phase of the reconstruction algorithm, a naïve solver based on an exhaustive search of all the use substitutions would be unusable, since the number of use variables is typically large, even in small processes. Incidentally, note that completion contributes significantly to this number, since it generates fresh use variables for all the instantiated channel types.

There are two simple yet effective strategies that can be used for speeding up the search of a particular use substitution (both have been implemented in the prototype). The first strategy is based on the observation that, although the set of use variables can be large, it can often be partitioned into many independent subsets. Finding partitions is easy: two variables ϱ_1 and ϱ_2 are *related* in \mathcal{C} if $\mathcal{C} \Vdash U \hat{=} V$ and ϱ_1, ϱ_2 occur in $U \hat{=} V$ (regardless of where ϱ_1 and ϱ_2 occur exactly). The dependencies between variables induce a partitioning of the use constraints such that the use variables occurring in the constraints of a partition are all related among them, and are not related with any other use variable occurring in a use constraint outside the partition. Once the partitioning of use constraints has been determined, each partition can be solved independently of the others.

The second strategy is based on the observation that many use constraints have the form $\varrho \hat{=} U$ where ϱ does not occur in U . In this case, the value of ϱ is in fact determined by U . So, U can be substituted in place of all the occurrences of ϱ in a given set of use constraints and, once a substitution is found for the use variables in the set of use constraints with the substitution, the substitution for ϱ can be determined by simply evaluating U under such substitution.

6.3. Pair splitting versus pair projection. It is usually the case that linearly typed languages provide a dedicated construct for splitting pairs (a notable exception is [14]). The language introduced in [23, Chapter 1], for example, has an expression form

$$\text{split } e \text{ as } x, y \text{ in } f$$

that evaluates e to a pair, binds the first and second component of the pair respectively to the variables x and y , and then evaluates f . At the same time, no pair projection primitives are usually provided. This is because in most linear type systems linear values “contaminate” with linearity the composite data structures in which they occur: for example, a pair containing linear values is itself a linear value and can only be used once, whereas for extracting *both* components of a pair using the projections one would have to project the pair *twice*, once using `fst` and one more time using `snd`. For this reason, the `split` construct becomes the only way to use linear pairs without violating linearity, as it grants access to both components of a pair but accessing the pair only once.

The process language we used in an early version of this article [20] provided a `split` construct for splitting pairs and did not have the projections `fst` and `snd`. In fact, the ability to use `fst` and `snd` without violating linearity constraints in our type system was pointed out by a reviewer of [20] and in this article we have decided to promote projections as the sole mechanism for accessing pair components. Notwithstanding this, there is a

practical point in favor of `split` when considering an actual implementation of the type system. Indeed, the pair projection rules [I-FST] and [I-SND] are among the few that generate constraints of the form $\text{un}(\alpha)$ for some type variable α . In the case of [I-FST] and [I-SND], the unlimited type variable stands for the component of the pair that is discarded by the projection. For instance, we can derive

$$\frac{x : \beta_1 \blacktriangleright x : \beta_1; \emptyset}{\text{fst}(x) : \alpha_1 \blacktriangleright x : \beta_1; \{\beta_1 \hat{=} \alpha_1 \times \gamma_1, \text{un}(\gamma_1)\}} \quad \frac{x : \beta_2 \blacktriangleright x : \beta_2; \emptyset}{\text{snd}(x) : \alpha_2 \blacktriangleright x : \beta_2; \{\beta_2 \hat{=} \alpha_2 \times \gamma_2, \text{un}(\gamma_2)\}}$$

$$\frac{(\text{fst}(x), \text{snd}(x)) : \alpha_1 \times \alpha_2 \blacktriangleright x : \alpha; \{\alpha \hat{=} \beta_1 + \beta_2, \beta_1 \hat{=} \alpha_1 \times \gamma_1, \beta_2 \hat{=} \alpha_2 \times \gamma_2, \text{un}(\gamma_1), \text{un}(\gamma_2)\}}{\text{fst}(x), \text{snd}(x) : \alpha_1 \times \alpha_2 \blacktriangleright x : \alpha; \{\alpha \hat{=} \beta_1 + \beta_2, \beta_1 \hat{=} \alpha_1 \times \gamma_1, \beta_2 \hat{=} \alpha_2 \times \gamma_2, \text{un}(\gamma_1), \text{un}(\gamma_2)\}}$$

and we observe that γ_1 and γ_2 are examples of those type variables for which only structural information is known, but no definition is present in the constraint set. Compare this with a hypothetical derivation concerning a splitting construct (for expressions)

$$\frac{x_1 : \alpha_1 \blacktriangleright x_1 : \alpha_1; \emptyset \quad x_2 : \alpha_2 \blacktriangleright x_2 : \alpha_2; \emptyset}{x : \alpha \blacktriangleright x : \alpha; \emptyset \quad (x_1, x_2) : \alpha_1 \times \alpha_2 \blacktriangleright x_1 : \alpha_1, x_2 : \alpha_2; \emptyset}$$

$$\frac{\text{split } x \text{ as } x_1, x_2 \text{ in } (x_1, x_2) : \alpha_1 \times \alpha_2 \blacktriangleright x : \alpha; \{\alpha \hat{=} \alpha_1 \times \alpha_2\}}{\text{split } x \text{ as } x_1, x_2 \text{ in } (x_1, x_2) : \alpha_1 \times \alpha_2 \blacktriangleright x : \alpha; \{\alpha \hat{=} \alpha_1 \times \alpha_2\}}$$

producing a much smaller constraint set which, in addition, is free from $\text{un}(\cdot)$ constraints and includes a definition for α . The constraint set obtained from the second derivation is somewhat easier to solve, if only because it requires no completion, meaning fewer use variables to generate and fewer chances of stumbling on the approximated solution of use constraints (Example 5.11).

Incidentally we observe, somehow surprisingly, that the two constraint sets are not exactly equivalent. In particular, the constraint set obtained from the first derivation admits a solution containing the substitutions

$$\{\alpha \mapsto [\text{int}]^{\omega,0} \times [\text{int}]^{0,\omega}, \alpha_1 \mapsto [\text{int}]^{1,0}, \alpha_2 \mapsto [\text{int}]^{0,1}\}$$

whereas in the second derivation, if we fix α as in the substitution above, we can only have

$$\{\alpha \mapsto [\text{int}]^{\omega,0} \times [\text{int}]^{0,\omega}, \alpha_1 \mapsto [\text{int}]^{\omega,0}, \alpha_2 \mapsto [\text{int}]^{0,\omega}\}$$

meaning that, using projections, it is possible to *extract* from a pair only the needed capabilities, provided that what remains unused has an unlimited type. On the contrary, `split` always extracts the full set of capabilities from each component of the pair.

In conclusion, in spite of the features of the type system we argue that it is a good idea to provide both pair projections and pair splitting, and that pair splitting should be preferred whenever convenient to use.

7. EXAMPLES

In this section we discuss three more elaborate examples that highlight the features of our type reconstruction algorithm. For better clarity, in these examples we extend the language with triples, boolean values, conditional branching, arithmetic and relational operators, OCaml-like polymorphic variants [16, Chapter 4], and a more general form of pattern matching. All these extensions can be easily accommodated or encoded in the language presented in Section 2 and are supported by the prototype implementation of the reconstruction algorithm.

$?t.T$ (input a message of type t , then use the channel according to T) or $!t.T$ (output a message of type t , then use the channel according to T) and possibly others for describing terminated protocols and protocols with branching structure. By considering also recursive session types (as done, *e.g.*, in [5]), or by taking the regular trees over such constructors (as we have done for our type language in this paper), it is possible to describe potentially infinite protocols. For instance, the infinite regular tree T satisfying the equality

$$T = !\mathbf{int}.\mathbf{?bool}.T$$

describes the protocol followed by a process that alternates outputs of integers and inputs of booleans on a session channel, whereas the infinite regular tree S satisfying the equality

$$S = \mathbf{?int}.\mathbf{!bool}.S$$

describes the protocol followed by a process that alternates inputs of integers and outputs of booleans. According to the conventional terminology, T and S above are *dual* of each other: each action described in T (like the output of a message of type \mathbf{int}) is matched by a corresponding co-action in S (like the input of a message of type \mathbf{int}). This implies that two processes that respectively follow the protocols T and S when using the *same* session channel can interact without errors: when one process sends a message of type t on the channel, the other process is ready to receive a message of the same type from the channel. Two such processes are those yielded by the outputs $\mathbf{foo!}c$ and $\mathbf{bar!}c$ below:

$$\begin{aligned} &*\mathbf{foo?}(x).x!\mathbf{random}.x?(_).\mathbf{foo!}x \\ &|\mathbf{*bar?}(y).y?(n).y!(n \bmod 2).\mathbf{bar!}y \\ &|\mathbf{new } c \mathbf{ in } (\mathbf{foo!}c | \mathbf{bar!}c) \end{aligned}$$

It is easy to trace a correspondence of the actions described by T with the operations performed on x , and of the actions described by S with the operations performed on y . Given that x and y are instantiated with the same channel c , and given the duality that relates T and S , this process exhibits no communication errors even if the *same* channel c is exchanging messages of *different* types (\mathbf{int} or \mathbf{bool}). For this reason, c is not a linear channel and the above process is ill typed according to our typing discipline. However, as discussed in [13, 3, 2], binary sessions and binary session types can be *encoded* in the linear π -calculus using a continuation passing style. The key idea of the encoding is that each communication in a session is performed on a distinct linear channel, and the exchanged message carries, along with the actual payload, a continuation channel on which the rest of the conversation takes place. According to this intuition, the process above is encoded in the linear π -calculus as the term:

$$\begin{aligned} &*\mathbf{foo?}(x).\mathbf{new } a \mathbf{ in } (x!(\mathbf{random},a) | a?(_,x').\mathbf{foo!}x') \\ &|\mathbf{*bar?}(y).y?(n,y').\mathbf{new } b \mathbf{ in } (y'!(n \bmod 2,b) | \mathbf{bar!}b) \\ &|\mathbf{new } c \mathbf{ in } (\mathbf{foo!}c | \mathbf{bar!}c) \end{aligned}$$

where a , b , and c are all linear channels (with possibly different types) used for exactly one communication. The encoding of processes using (binary) sessions into the linear π -calculus induces a corresponding encoding of session types into linear channel types. In particular, input and output session types are encoded according to the laws

$$\begin{aligned} \llbracket ?t.T \rrbracket &= \llbracket t \times \llbracket T \rrbracket \rrbracket^{1,0} \\ \llbracket !t.T \rrbracket &= \llbracket t \times \llbracket T \rrbracket \rrbracket^{0,1} \end{aligned} \tag{7.1}$$

where we use \overline{T} to denote the dual protocol of T . Such encoding is nothing but the coinductive extension of the one described in [3] to infinite protocols. Note that in $[[!t.T]]$, the type of the continuation channel is the encoding of the *dual* of T . This is because the transmitted continuation will be used *by the receiver process* in a complementary fashion with respect to T , which instead describes the continuation of the protocol from the viewpoint of the sender. As an example, the protocols T and S above can be respectively encoded as the types t and s that satisfy the equalities

$$t = [\text{int} \times [\text{bool} \times s]^{0,1}]^{0,1} \quad s = [\text{int} \times [\text{bool} \times s]^{0,1}]^{1,0}$$

It turns out that these are the types that our type reconstruction algorithm associates with x and y . This is not a coincidence, for essentially three reasons: (1) the encoding of a well-typed process making use of binary sessions is always a well-typed process in the linear π -calculus [3, 2], (2) our type reconstruction algorithm is complete (Theorem 4.3), and (3) it can identify a channel as linear when it is used for one communication only (Section 5.4). The upshot is that, once the types t and s have been reconstructed, the protocols T and S can be obtained by a straightforward procedure that “decodes” t and s using the inverse of the transformation sketched by the equations (7.1). There is a technical advantage of such rather indirect way of performing session type reconstruction. Duality accounts for a good share of the complexity of algorithms that reconstruct session types directly [17]. However, as the authors of [3] point out, the notion of duality that relates T and S – and that *globally* affects their structure – boils down to a *local* swapping of uses in the topmost channel types in t and s . This is a general property of the encoding that has important practical implications: the hard work is carried over during type reconstruction for the linear π -calculus, where there is no duality to worry about; once such phase is completed, session types can be obtained from linear channel types with little effort.

We have equipped the prototype implementation of the type reconstruction algorithm with a flag that decodes linear channel types into session types (the decoding procedure accounts for a handful of lines of code). In this way, the tool can be used for inferring the communication protocol of processes encoded in the linear π -calculus. Since the type reconstruction algorithm supports infinite and disjoint sum types, both infinite protocols and protocols with branches can be inferred. Examples of such processes, like for instance the server for mathematical operations described in [5], are illustrated on the home page of the tool and in its source archive. ■

Example 7.3. In this example we motivate the requirement expressed in the rules [T-NEW] and [I-NEW] imposing that the type of restricted channels should have the same use in its input/output use slots. To this aim, consider the process below

```
*filter?(a,b).a?(n,c).if n ≥ 0 then new d in (b!(n,d1) | filter!(c,d2))
    else filter!(c,b)
```

which filters numbers received from channel a and forwards the non-negative ones on channel b . Each number n comes along with a continuation channel c from which the next number in the stream will be received. Symmetrically, any message sent on b includes a continuation d on which the next non-negative number will be sent. For convenience, we distinguish d bound by `new` from the two rightmost occurrences d_1 and d_2 of d .

For this process the reconstruction algorithm infers the type

$$\text{filter} : [t \times [\text{int} \times t]^{0,1}]^{\omega,\omega} \tag{7.2}$$

where t is the type that satisfies the equality $t = [\mathbf{int} \times t]^{1,0}$ meaning that d_1 and d_2 are respectively assigned the types t and $[\mathbf{int} \times t]^{0,1}$ and overall d has type $t + [\mathbf{int} \times t]^{0,1} = [\mathbf{int} \times t]^{1,0} + [\mathbf{int} \times t]^{0,1} = [\mathbf{int} \times t]^{1,1}$. The reason why d_2 has type $[\mathbf{int} \times t]^{0,1}$, namely that d_2 is used for an output operation, is clear, since d_2 must have the same type as b and b is indeed used for an output operation in the body of `filter`. However, in the whole process there is no explicit evidence that d_1 will be used for an input operation, and the input use 1 in its type $t = [\mathbf{int} \times t]^{1,0}$ is deduced “by subtraction”, as we have discussed in the informal overview at the beginning of Section 5.

If we do *not* impose the constraint that restricted (linear) channel should have the same input/output use, we can find a more precise solution that determines for `filter` the type

$$\mathbf{filter} : [t \times [\mathbf{int} \times s]^{0,1}]^{\omega,\omega} \quad (7.3)$$

where s is the type that satisfies the equality $s = [\mathbf{int} \times s]^{0,0}$. According to (7.3), d_1 is assigned the type s saying that no operation will ever be performed on it. This phenomenon is a consequence of the fact that, when we apply the type reconstruction algorithm on an isolated process, like `filter` above, which is never invoked, the reconstruction algorithm has only a partial view of the behavior of the process on the channel it creates. For extruded channels like d , in particular, the algorithm is unable to infer any direct use. We argue that the typing (7.3) renders `filter` a useless process from which it is not possible to receive any message, unless `filter` is typed along with the rest of the program that invokes it. But this latter strategy prevents *de facto* the modular application of the reconstruction algorithm to the separate constituents of a program.

The typing (7.2) is made possible by the completion phase (Section 5), which is an original feature of our type reconstruction algorithm. The prototype implementation of the algorithm provides a flag that disables the constraint on equal uses in [I-NEW] allowing experimentation of the behavior of the algorithm on examples like this one. ■

8. CONCLUDING REMARKS

Previous works on the linear π -calculus either do not treat composite types [15, 10] or are based on an interpretation of linearity that limits data sharing and parallelism [8, 9]. Type reconstruction for recursive or, somewhat equivalently, infinite types has also been neglected, despite the key role played by these types for describing structured data (lists, trees, etc.) and structured interactions [2]. In this work we have extended the linear π -calculus with both composite and infinite types and have adopted a more relaxed attitude towards linearity that fosters data sharing and parallelism while maintaining the availability of a type reconstruction algorithm. The extension is a very natural one, as witnessed by the fact that our type system uses essentially the same rules of previous works, the main novelty being a different type combination operator. This small change has nonetheless non-trivial consequences on the reconstruction algorithm, which must reconcile the propagation of constraints across composite types and the impossibility to rely on plain type unification: different occurrences of the same identifier may be assigned different types and types may be infinite. Our extension also gives renewed relevance to types like $[t]^{0,0}$. In previous works these types were admitted but essentially useless: channels with such types could only be passed around in messages without actually ever being used. That is, they could be erased without affecting processes. In our type system, it is the existence of these types

that enables the sharing of structured data (see the decomposition of t_{list} into t_{even} and t_{odd} in Section 1).

Binary sessions [6, 7] can be encoded into the linear π -calculus [13, 3]. Thus, we indirectly provide a complete reconstruction algorithm for possibly infinite, higher-order, binary session types. As shown in [17], direct session type reconstruction poses two major technical challenges: on the one hand, the necessity to deal with dual types; on the other hand, the fact that subtyping must be taken into account for that is the only way to properly handle selections in conditionals. Interestingly, both complications disappear when session types are encoded in the linear π -calculus: duality simply turns into swapping the input/output use annotations in channel types [3], whereas selections become outputs of variant data types which can be dealt with using conventional techniques based on unification [16].

To assess the feasibility of the approach, we have implemented the type reconstruction algorithm in a tool for the static analysis of π -calculus processes. Given that even simple processes generate large constraint sets, the prototype has been invaluable for testing the algorithm at work on non-trivial examples. The reconstruction described in this article is only the first step for more advanced forms of analysis, such as those for reasoning on deadlocks and locks [19]. We have extended the tool in such a way that subsequent analyses can be plugged on top of the reconstruction algorithm for linear channels [21].

Structural subtyping and polymorphism are two natural developments of our work. The former has already been considered in [9], but it is necessary to understand how it integrates with our notion of type combination and how it affects constraint generation and resolution. Polymorphism makes sense for unlimited channels only (there is little point in having polymorphic linear channels, since they can only be used once anyway). Nevertheless, support for polymorphism is not entirely trivial, since some type variables may need to be restricted to unlimited types. For example, the channel `first` in the process `*first?(x, y).y!fst(x)` would have type $\forall\alpha.\forall\beta.\text{un}(\beta) \Rightarrow [(\alpha \times \beta) \times [\alpha]^{0,1}]^{\omega,0}$.

Acknowledgements. The author is grateful to the anonymous reviewers whose numerous questions, detailed comments and suggestions have significantly contributed to improving both content and presentation of this article. The author is also grateful to Naoki Kobayashi for his comments on an earlier version of the article.

REFERENCES

- [1] B. Courcelle. Fundamental properties of infinite trees. *Theor. Comp. Sci.*, 25:95–169, 1983.
- [2] O. Dardha. Recursive session types revisited. In *BEAT'14*, 2014.
- [3] O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In *PPDP'12*, pages 139–150. ACM, 2012.
- [4] R. Demangeon and K. Honda. Full abstraction in a subtyped pi-calculus with linear types. In *CONCUR'11*, LNCS 6901, pages 280–296. Springer, 2011.
- [5] S. J. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005.
- [6] K. Honda. Types for dyadic interaction. In *CONCUR'93*, LNCS 715, pages 509–523. Springer, 1993.
- [7] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, LNCS 1381, pages 122–138. Springer, 1998.
- [8] A. Igarashi. Type-based analysis of usage of values for concurrent programming languages, 1997. Available at <http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/papers/>.
- [9] A. Igarashi and N. Kobayashi. Type-based analysis of communication for concurrent programming languages. In *SAS'97*, LNCS 1302, pages 187–201. Springer, 1997.

- [10] A. Igarashi and N. Kobayashi. Type Reconstruction for Linear π -Calculus with I/O Subtyping. *Inf. and Comp.*, 161(1):1–44, 2000.
- [11] N. Kobayashi. Quasi-linear types. In *POPL'99*, pages 29–42. ACM, 1999.
- [12] N. Kobayashi. A type system for lock-free processes. *Inf. and Comp.*, 177(2):122–159, 2002.
- [13] N. Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, LNCS 2757, pages 439–453. Springer, 2002. Extended version at <http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf>.
- [14] N. Kobayashi. A new type system for deadlock-free processes. In *CONCUR'06*, LNCS 4137, pages 233–247. Springer, 2006.
- [15] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999.
- [16] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system release 4.01, 2013. Available at <http://caml.inria.fr/pub/docs/manual-ocaml-4.01/index.html>.
- [17] L. G. Mezzina. How to infer finite session types in a calculus of services and sessions. In *COORDINATION'08*, LNCS 5052, pages 216–231. Springer, 2008.
- [18] U. Nestmann and M. Steffen. Typing confluence. In *FMICS'97*, pages 77–101, 1997. Also available as report ERCIM-10/97-R052, European Research Consortium for Informatics and Mathematics, 1997.
- [19] L. Padovani. Deadlock and Lock Freedom in the Linear π -Calculus. In *CSL-LICS'14*, pages 72:1–72:10. ACM, 2014.
- [20] L. Padovani. Type reconstruction for the linear π -calculus with composite and equi-recursive types. In *FoSSaCS'14*, LNCS 8412, pages 88–102. Springer, 2014.
- [21] L. Padovani, T.-C. Chen, and A. Tosatto. Type Reconstruction Algorithms for Deadlock-Free and Lock-Free Linear π -Calculi. In *COORDINATION'15*, LNCS 9037, pages 83–98. Springer, 2015.
- [22] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
- [23] B. C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.
- [24] D. Sangiorgi and D. Walker. *The Pi-Calculus - A theory of mobile processes*. Cambridge University Press, 2001.
- [25] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *FPCA'95*, pages 1–11, 1995.

APPENDIX A. SUPPLEMENT TO SECTION 3

To prove Theorem 3.8 we need a series of standard auxiliary results, including weakening (Lemma A.1) and substitution (Lemma A.2) for both expressions and processes.

Lemma A.1 (weakening). *The following properties hold:*

- (1) *If $\Gamma \vdash e : t$ and $\text{un}(\Gamma')$ and $\Gamma + \Gamma'$ is defined, then $\Gamma + \Gamma' \vdash e : t$.*
- (2) *If $\Gamma \vdash P$ and $\text{un}(\Gamma')$ and $\Gamma + \Gamma'$ is defined, then $\Gamma + \Gamma' \vdash P$.*

Proof. Both items are proved by a standard induction on the typing derivation. In case (2) we assume, without loss of generality, that $\text{bn}(P) \cap \text{dom}(\Gamma) = \emptyset$ (recall that we identify processes modulo renaming of bound names). □

Lemma A.2 (substitution). *Let $\Gamma_1 \vdash v : t$. The following properties hold:*

- (1) *If $\Gamma_2, x : t \vdash e : s$ and $\Gamma_1 + \Gamma_2$ is defined, then $\Gamma_1 + \Gamma_2 \vdash e\{v/x\} : s$.*
- (2) *If $\Gamma_2, x : t \vdash P$ and $\Gamma_1 + \Gamma_2$ is defined, then $\Gamma_1 + \Gamma_2 \vdash P\{v/x\}$.*

Proof. The proofs are standard, except for the following property of the type system: $\text{un}(t)$ implies $\text{un}(\Gamma_1)$, which can be easily proved by induction on the derivation of $\Gamma_1 \vdash v : t$. □

Next is type preservation under structural pre-congruence.

Lemma A.3. *If $\Gamma \vdash P$ and $P \preceq Q$, then $\Gamma \vdash Q$.*

Proof. We only show the case in which a replicated process is expanded. Assume $P = *P' \preceq *P' \mid P' = Q$. From the hypothesis $\Gamma \vdash P$ and [T-REP] we deduce $\Gamma \vdash P'$ and $\text{un}(\Gamma)$. By definition of unlimited environment (see Definition 3.7) we have $\Gamma = \Gamma + \Gamma$. We conclude $\Gamma \vdash Q$ with an application of [T-PAR]. \square

Lemma A.4. *If $\Gamma \xrightarrow{\ell} \Gamma'$ and $\Gamma + \Gamma''$ is defined, then $\Gamma + \Gamma'' \xrightarrow{\ell} \Gamma' + \Gamma''$.*

Proof. Easy consequences of the definition of $\xrightarrow{\ell}$ on type environments. \square

We conclude with type preservation for expressions and subject reduction for processes.

Lemma A.5. *Let $\Gamma \vdash e : t$ and $e \downarrow v$. Then $\Gamma \vdash v : t$.*

Proof. By induction on $e \downarrow v$ using the hypothesis that e is well typed. \square

Theorem 3.8. *Let $\Gamma \vdash P$ and $P \xrightarrow{\ell} Q$. Then $\Gamma' \vdash Q$ for some Γ' such that $\Gamma \xrightarrow{\ell} \Gamma'$.*

Proof. By induction on the derivation of $P \xrightarrow{\ell} Q$ and by cases on the last rule applied. We only show a few interesting cases; the others are either similar or simpler.

[R-COMM] Then $P = e_1 !f \mid e_2?(x).R$ and $e_i \downarrow a$ for every $i = 1, 2$ and $f \downarrow v$ and $\ell = a$ and $Q = R\{v/x\}$. From [T-PAR] we deduce $\Gamma = \Gamma_1 + \Gamma_2$ where $\Gamma_1 \vdash e_1 !f$ and $\Gamma_2 \vdash e_2?(x).R$. From [T-OUT] we deduce $\Gamma_1 = \Gamma_{11} + \Gamma_{12}$ and $\Gamma_{11} \vdash e_1 : [t]^{2\kappa_1, 1+\kappa_2}$ and $\Gamma_{12} \vdash f : t$. From [T-IN] we deduce $\Gamma_2 = \Gamma_{21} + \Gamma_{22}$ and $\Gamma_{21} \vdash e_2 : [s]^{1+\kappa_3, 2\kappa_4}$ and $\Gamma_{22}, x : s \vdash R$. From Lemma A.5 we have $\Gamma_{11} \vdash a : [t]^{2\kappa_1, 1+\kappa_2}$ and $\Gamma_{12} \vdash v : t$ and $\Gamma_{21} \vdash a : [s]^{1+\kappa_3, 2\kappa_4}$. Also, since $\Gamma_{11} + \Gamma_{21}$ is defined, it must be the case that $t = s$. Note that $1 + \kappa_2 = 1 + 2\kappa_2$ and $1 + \kappa_3 = 1 + 2\kappa_3$. Hence, from [T-NAME] we deduce that $\Gamma_{11} = \Gamma'_{11}, a : [t]^{2\kappa_1, 1+\kappa_2} = (\Gamma'_{11}, a : [t]^{2\kappa_1, 2\kappa_2}) + a : [t]^{0,1}$ and $\Gamma_{21} = \Gamma'_{21}, a : [t]^{1+\kappa_3, 2\kappa_4} = (\Gamma'_{21}, a : [t]^{2\kappa_3, 2\kappa_4}) + a : [t]^{1,0}$ for some unlimited Γ'_{11} and Γ'_{21} . Let $\Gamma''_{11} \stackrel{\text{def}}{=} \Gamma'_{11}, a : [t]^{2\kappa_1, 2\kappa_2}$ and $\Gamma''_{12} \stackrel{\text{def}}{=} \Gamma'_{21}, a : [t]^{2\kappa_3, 2\kappa_4}$ and observe that Γ''_{11} and Γ''_{12} are also unlimited. From Lemma A.2 we deduce $\Gamma_{12} + \Gamma_{22} \vdash R\{v/x\}$. Take $\Gamma' = \Gamma''_{11} + \Gamma_{12} + \Gamma''_{12} + \Gamma_{22}$. From Lemma A.1 we deduce $\Gamma' \vdash Q$ and we conclude by observing that $\Gamma \xrightarrow{a} \Gamma'$ thanks to Lemma A.4.

[R-CASE] Then $P = \text{case } e \{i(x_i) \Rightarrow P_i\}_{i=\text{inl}, \text{inr}}$ and $e \downarrow k(v)$ for some $k \in \{\text{inl}, \text{inr}\}$ and $\ell = \tau$ and $Q = P_k\{v/x_k\}$. From [T-CASE] we deduce that $\Gamma = \Gamma_1 + \Gamma_2$ and $\Gamma_1 \vdash e : t_{\text{inl}} \oplus t_{\text{inr}}$ and $\Gamma_2, x : t_k \vdash P_k$. From Lemma A.5 and either [T-INL] or [T-INR] we deduce $\Gamma_1 \vdash v : t_k$. We conclude $\Gamma \vdash P_k\{v/x_k\}$ by Lemma A.2.

[R-PAR] Then $P = P_1 \mid P_2$ and $P_1 \xrightarrow{\ell} P'_1$ and $Q = P'_1 \mid P_2$. From [T-PAR] we deduce $\Gamma = \Gamma_1 + \Gamma_2$ and $\Gamma_i \vdash P_i$. By induction hypothesis we deduce $\Gamma'_1 \vdash P'_1$ for some Γ'_1 such that $\Gamma_1 \xrightarrow{\ell} \Gamma'_1$. By Proposition A.4 we deduce that $\Gamma \xrightarrow{\ell} \Gamma'_1 + \Gamma_2$. We conclude $\Gamma' \vdash Q$ by taking $\Gamma' = \Gamma'_1 + \Gamma_2$. \square

APPENDIX B. SUPPLEMENT TO SECTION 4

First of all we prove two technical lemmas that explain the relationship between the operators \sqcup and \sqcap used by the constraint generation rules (Table 7) and type environment combination $+$ and equality used in the type rules (Table 4).

Lemma B.1. *If $\Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \mathcal{C}$ and σ is a solution for \mathcal{C} covering Δ , then $\sigma\Delta = \sigma\Delta_1 + \sigma\Delta_2$.*

Proof. By induction on the derivation of $\Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \mathcal{C}$ and by cases on the last rule applied. We have two cases:

$\boxed{\text{dom}(\Delta_1) \cap \text{dom}(\Delta_2) = \emptyset}$ Then $\Delta = \Delta_1, \Delta_2$ and we conclude $\sigma\Delta = \sigma\Delta_1, \sigma\Delta_2 = \sigma\Delta_1 + \sigma\Delta_2$.

$\boxed{\Delta_1 = \Delta'_1, u : \mathsf{T}$ and $\Delta_2 = \Delta'_2, u : \mathsf{S}}$ Then $\Delta'_1 \sqcup \Delta'_2 \rightsquigarrow \Delta'; \mathcal{C}'$ and $\Delta = \Delta', u : \alpha$ and $\mathcal{C} = \mathcal{C}' \cup \{\alpha \hat{=} \mathsf{T} + \mathsf{S}\}$ for some α . Since σ is a solution for \mathcal{C} , we deduce $\sigma(\alpha) = \sigma\mathsf{T} + \sigma\mathsf{S}$. By induction hypothesis we deduce $\sigma\Delta' = \sigma\Delta'_1 + \sigma\Delta'_2$. We conclude $\sigma\Delta = \sigma\Delta', u : \sigma(\alpha) = \sigma\Delta', u : \sigma\mathsf{T} + \sigma\mathsf{S} = (\sigma\Delta'_1 + \sigma\Delta'_2), u : \sigma\mathsf{T} + \sigma\mathsf{S} = \sigma\Delta_1 + \sigma\Delta_2$. \square

Lemma B.2. *If $\Delta_1 \sqcap \Delta_2 \rightsquigarrow \Delta; \mathcal{C}$ and σ is a solution for \mathcal{C} covering Δ , then $\sigma\Delta = \sigma\Delta_1 = \sigma\Delta_2$.*

Proof. Straightforward consequence of the definition of $\Delta_1 \sqcap \Delta_2 \rightsquigarrow \Delta; \mathcal{C}$. \square

The correctness of constraint generation is proved by the next two results.

Lemma B.3. *If $e : \mathsf{T} \blacktriangleright \Delta; \mathcal{C}$ and σ is a solution for \mathcal{C} covering Δ , then $\sigma\Delta \vdash e : \sigma\mathsf{T}$.*

Proof. By induction on the derivation of $e : \mathsf{T} \blacktriangleright \Delta; \mathcal{C}$ and by cases on the last rule applied. We only show two significant cases.

$\boxed{[\text{I-NAME}]}$ Then $e = u$ and $\mathsf{T} = \alpha$ fresh and $\Delta = u : \alpha$ and $\mathcal{C} = \emptyset$. We have $\sigma\Delta = u : \sigma(\alpha)$ and $\sigma\mathsf{T} = \sigma(\alpha)$, hence we conclude $\sigma\Delta \vdash e : \sigma\mathsf{T}$.

$\boxed{[\text{I-PAIR}]}$ Then $e = (e_1, e_2)$ and $\mathsf{T} = \mathsf{T}_1 \times \mathsf{T}_2$ and $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3$ where $\Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \mathcal{C}_3$ and $e_i : \mathsf{T}_i \blacktriangleright \Delta_i; \mathcal{C}_i$ for $i = 1, 2$. We know that σ is a solution for \mathcal{C}_i for all $i = 1, 2, 3$. By induction hypothesis we deduce $\sigma\Delta_i \vdash e : \sigma\mathsf{T}_i$ for $i = 1, 2$. From Lemma B.1 we obtain $\sigma\Delta = \sigma\Delta_1 + \sigma\Delta_2$. We conclude with an application of $[\text{T-PAIR}]$. \square

Theorem 4.2. *If $P \blacktriangleright \Delta; \mathcal{C}$ and σ is a solution for \mathcal{C} that covers Δ , then $\sigma\Delta \vdash P$.*

Proof. By induction on the derivation of $P \blacktriangleright \Delta; \mathcal{C}$ and by cases on the last rule applied.

$\boxed{[\text{I-IDLE}]}$ Then $P = \text{idle}$ and $\Delta = \emptyset$ and $\mathcal{C} = \emptyset$. We conclude with an application of $[\text{T-IDLE}]$.

$\boxed{[\text{I-IN}]}$ Then $P = e?(x).Q$ and $e : \mathsf{T} \blacktriangleright \Delta_1; \mathcal{C}_1$ and $Q \blacktriangleright \Delta_2, x : \mathsf{S}; \mathcal{C}_2$ and $\Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \mathcal{C}_3$ and $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\mathsf{T} \hat{=} [\mathsf{S}]^{1+\varrho_1, 2\varrho_2}\}$. By Lemma B.3 we deduce $\sigma\Delta_1 \vdash e : \sigma\mathsf{T}$. By induction hypothesis we deduce $\sigma\Delta_2, x : \sigma\mathsf{S} \vdash Q$. By Lemma B.1 we deduce $\sigma\Delta = \sigma\Delta_1 + \sigma\Delta_2$. From the hypothesis that σ is a solution for \mathcal{C} we know $\sigma\mathsf{T} = [\sigma\mathsf{S}]^{1+\sigma(\varrho_1), 2\sigma(\varrho_2)}$. We conclude with an application of $[\text{T-IN}]$.

$\boxed{[\text{I-OUT}]}$ Then $P = e!f$ and $e : \mathsf{T} \blacktriangleright \Delta_1; \mathcal{C}_1$ and $f : \mathsf{S} \blacktriangleright \Delta_2; \mathcal{C}_2$ and $\Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \mathcal{C}_3$ and $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\mathsf{T} \hat{=} [\mathsf{S}]^{2\varrho_1, 1+\varrho_2}\}$. By Lemma B.3 we deduce $\sigma\Delta_1 \vdash e : \sigma\mathsf{T}$ and $\sigma\Delta_2 \vdash f : \sigma\mathsf{S}$. By Lemma B.1 we deduce $\sigma\Delta = \sigma\Delta_1 + \sigma\Delta_2$. From the hypothesis that σ is a solution for \mathcal{C} we know $\sigma\mathsf{T} = [\sigma\mathsf{S}]^{2\sigma(\varrho_1), 1+\sigma(\varrho_2)}$. We conclude with an application of $[\text{T-OUT}]$.

[I-PAR] Then $P = P_1 | P_2$ and $P_i \blacktriangleright \Delta_i; \mathcal{C}_i$ for $i = 1, 2$ and $\Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \mathcal{C}_3$ and $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3$. By induction hypothesis we deduce $\sigma\Delta_i \vdash P_i$ for $i = 1, 2$. By Lemma B.1 we deduce $\sigma\Delta = \sigma\Delta_1 + \sigma\Delta_2$. We conclude with an application of [T-PAR].

[I-REP] Then $P = *Q$ and $Q \blacktriangleright \Delta'; \mathcal{C}_1$ and $\Delta' \sqcup \Delta' \rightsquigarrow \Delta; \mathcal{C}_2$ and $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$. By induction hypothesis we deduce $\sigma\Delta' \vdash Q$. By Lemma B.1 we deduce $\sigma\Delta = \sigma\Delta' + \sigma\Delta'$. By Definition 3.7 we know that $\text{un}(\sigma\Delta)$ holds. Furthermore, $\sigma\Delta' + \sigma\Delta$ is defined. By Lemma A.1 and Definition 3.5 we deduce $\sigma\Delta \vdash Q$. We conclude with an application of [T-REP].

[I-NEW] Then $P = \text{new } a \text{ in } Q$ and $Q \blacktriangleright \Delta, a : \top; \mathcal{C}'$ and $\mathcal{C} = \mathcal{C}' \cup \{\top \hat{=} [\alpha]^{e, \ell}\}$. By induction hypothesis we deduce $\sigma\Delta, a : \sigma\top \vdash Q$. Since σ is a solution for \mathcal{C}' we know that $\sigma\top = [\sigma(\alpha)]^{\sigma(e), \sigma(\ell)}$. We conclude with an application of [T-NEW].

[I-CASE] Then $P = \text{case } e \{i(x_i) \Rightarrow P_i\}_{i=\text{inl}, \text{inr}}$ and $e : t \blacktriangleright \Delta_1; \mathcal{C}_1$ and $P_i \blacktriangleright \Delta_i, x_i : \top_i; \mathcal{C}_i$ for $i = \text{inl}, \text{inr}$ and $\Delta_{\text{inl}} \sqcap \Delta_{\text{inr}} \rightsquigarrow \Delta_2; \mathcal{C}_2$ and $\Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \mathcal{C}_3$ and $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \mathcal{C}_{\text{inl}} \cup \mathcal{C}_{\text{inr}} \cup \{\top \hat{=} \top_{\text{inl}} \oplus \top_{\text{inr}}\}$. By Lemma B.3 we deduce $\sigma\Delta_1 \vdash e : \sigma\top$. By induction hypothesis we deduce $\sigma\Delta_i \vdash P_i$ for $i = \text{inl}, \text{inr}$. By Lemma B.2 we deduce $\sigma\Delta_{\text{inl}} = \sigma\Delta_{\text{inr}} = \sigma\Delta_2$. By Lemma B.1 we deduce $\sigma\Delta = \sigma\Delta_1 + \sigma\Delta_2$. Since σ is a solution for \mathcal{C} , we have $\sigma\top = \sigma\top_{\text{inl}} \oplus \sigma\top_{\text{inr}}$. We conclude with an application of [T-CASE].

[I-WEAK] Then $\Delta = \Delta', u : \alpha$ and $\mathcal{C} = \mathcal{C}' \cup \{\text{un}(\alpha)\}$ where α is fresh and $P \blacktriangleright \Delta'; \mathcal{C}'$. By induction hypothesis we deduce $\sigma\Delta' \vdash P$. Since σ is a solution for \mathcal{C}' we know that $\text{un}(\sigma(\alpha))$ holds. Since $u \notin \text{dom}(\Delta')$ we know that $\Delta'\sigma + u : \sigma(\alpha)$ is defined. By Lemma A.1(2) we conclude $\sigma\Delta', u : \sigma(\alpha) \vdash P$. \square

The next lemma relates once more \sqcup and type environment combination $+$. It is, in a sense, the inverse of Lemma B.1.

Lemma B.4. *If $\sigma\Delta_1 + \sigma\Delta_2$ is defined, then there exist Δ, \mathcal{C} , and $\sigma' \supseteq \sigma$ such that $\Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \mathcal{C}$ and σ' is a solution for \mathcal{C} that covers Δ .*

Proof. By induction on the maximum size of Δ_1 and Δ_2 . We distinguish two cases.

$\text{dom}(\Delta_1) \cap \text{dom}(\Delta_2) = \emptyset$ We conclude by taking $\Delta \stackrel{\text{def}}{=} \Delta_1, \Delta_2$ and $\mathcal{C} \stackrel{\text{def}}{=} \emptyset$ and $\sigma' \stackrel{\text{def}}{=} \sigma$ and observing that $\Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \emptyset$.

$\Delta_1 = \Delta'_1, u : \top$ and $\Delta_2 = \Delta'_2, u : \top$ Since $\sigma\Delta_1 + \sigma\Delta_2$ is defined, we know that $\sigma\Delta'_1 + \sigma\Delta'_2$ is defined as well and furthermore that $(\sigma\Delta_1 + \sigma\Delta_2)(u) = \sigma\top + \sigma\top$. By induction hypothesis we deduce that there exist Δ', \mathcal{C}' , and $\sigma'' \supseteq \sigma$ such that $\Delta'_1 \sqcup \Delta'_2 \rightsquigarrow \Delta'; \mathcal{C}'$ and σ'' is a solution for \mathcal{C}' that covers Δ' . Take $\Delta \stackrel{\text{def}}{=} \Delta', u : \alpha$ where α is fresh, $\mathcal{C} \stackrel{\text{def}}{=} \mathcal{C}' \cup \{\alpha \hat{=} \top + \top\}$ and $\sigma' \stackrel{\text{def}}{=} \sigma'' \cup \{\alpha \mapsto \sigma\top + \sigma\top\}$. We conclude observing that σ' is a solution for \mathcal{C} that covers Δ . \square

In order to prove the completeness of type reconstruction for expressions, we extend the reconstruction algorithm with one more weakening rule for expressions:

$$\frac{\text{[I-WEAK EXPR]} \quad e : \top \blacktriangleright \Delta; \mathcal{C}}{e : \top \blacktriangleright \Delta, u : \alpha; \mathcal{C} \cup \text{un}(\alpha)}$$

This rule is unnecessary as far as completeness is concerned, because there is already a weakening rule [I-WEAK] for processes that can be used to subsume it. However, [I-WEAK EXPR] simplifies both the proofs and the statements of the results that follow.

Lemma B.5. *If $\Gamma \vdash e : t$, then there exist \mathbb{T} , Δ , \mathcal{C} , and σ such that $e : \mathbb{T} \blacktriangleright \Delta; \mathcal{C}$ and σ is a solution for \mathcal{C} and $\Gamma = \sigma\Delta$ and $t = \sigma\mathbb{T}$.*

Proof. By induction on the derivation of $\Gamma \vdash e : t$ and by cases on the last rule applied. We only show two representative cases.

[T-NAME] Then $e = u$ and $\Gamma = \Gamma', u : t$ and $\text{un}(\Gamma')$. Let $\Gamma' = \{u_i : t_i\}_{i \in I}$. Take $\mathbb{T} \stackrel{\text{def}}{=} \alpha$ and $\Delta \stackrel{\text{def}}{=} \{u_i : \alpha_i\}_{i \in I}, u : \alpha$ and $\mathcal{C} \stackrel{\text{def}}{=} \{\text{un}(\alpha_i) \mid i \in I\}$ and $\sigma \stackrel{\text{def}}{=} \{\alpha_i \mapsto t_i\}_{i \in I} \cup \{\alpha \mapsto t\}$ where α and the α_i 's are all fresh type variables. Observe that $e : \mathbb{T} \blacktriangleright \Delta; \mathcal{C}$ by means of one application of [I-NAME] and as many applications of [I-WEAK EXPR] as the cardinality of I . We conclude observing that σ is a solution for \mathcal{C} and $\Gamma = \sigma\Delta$ and $t = \sigma\mathbb{T}$ by definition of σ .

[T-PAIR] Then $e = (e_1, e_2)$ and $\Gamma = \Gamma_1 + \Gamma_2$ and $t = t_1 \times t_2$ and $\Gamma_i \vdash e_i : t_i$ for $i = 1, 2$. By induction hypothesis we deduce that there exist $\mathbb{T}_i, \Delta_i, \mathcal{C}_i$, and σ_i solution for \mathcal{C}_i such that $e_i : \mathbb{T}_i \blacktriangleright \Delta_i; \mathcal{C}_i$ and $\Gamma_i = \sigma_i\Delta_i$ and $t_i = \sigma_i\mathbb{T}_i$ for $i = 1, 2$. Since the reconstruction algorithm always chooses fresh type variables, we also know that $\text{dom}(\sigma_1) \cap \text{dom}(\sigma_2) = \emptyset$. Take $\sigma' \stackrel{\text{def}}{=} \sigma_1 \cup \sigma_2$. We have that $\sigma'\Delta_1 + \sigma'\Delta_2 = \Gamma_1 + \Gamma_2$ is defined. Therefore, by Lemma B.4, we deduce that there exist Δ, \mathcal{C}_3 , and $\sigma \supseteq \sigma'$ such that $\Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \mathcal{C}_3$ and σ is a solution for \mathcal{C} that covers Δ . We conclude with an application of [I-PAIR] and taking $\mathbb{T} \stackrel{\text{def}}{=} \mathbb{T}_1 \times \mathbb{T}_2$ and $\mathcal{C} \stackrel{\text{def}}{=} \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3$. \square

Theorem 4.3. *If $\Gamma \vdash P$, then there exist Δ, \mathcal{C} , and σ such that $P \blacktriangleright \Delta; \mathcal{C}$ and σ is a solution for \mathcal{C} that covers Δ and $\Gamma = \sigma\Delta$.*

Proof. By induction on the derivation of $\Gamma \vdash P$ and by cases on the last rule applied. We only show a few cases, the others being analogous.

[T-IDLE] Then $P = \text{idle}$ and $\text{un}(\Gamma)$. Let $\Gamma = \{u_i : t_i\}_{i \in I}$. Take $\Delta \stackrel{\text{def}}{=} \{u_i : \alpha_i\}_{i \in I}$ and $\mathcal{C} \stackrel{\text{def}}{=} \{\text{un}(\alpha_i)\}_{i \in I}$ and $\sigma \stackrel{\text{def}}{=} \{\alpha_i \mapsto t_i\}_{i \in I}$ where the α_i 's are all fresh type variables. By repeated applications of [I-WEAK] and one application of [I-IDLE] we derive $\text{idle} \blacktriangleright \Delta; \mathcal{C}$. We conclude observing that σ is a solution for \mathcal{C} and $\Gamma = \sigma\Delta$.

[T-IN] Then $P = e?(x).Q$ and $\Gamma = \Gamma_1 + \Gamma_2$ and $\Gamma_1 \vdash e : [t]^{1+\kappa_1, 2\kappa_2}$ and $\Gamma_2, x : t \vdash Q$. By Lemma B.5 we deduce that there exist $\mathbb{T}, \Delta_1, \mathcal{C}_1$, and σ_1 solution for \mathcal{C}_1 such that $e : \mathbb{T} \blacktriangleright \Delta_1; \mathcal{C}_1$ and $\Gamma_1 = \sigma_1\Delta_1$ and $[t]^{1+\kappa_1, 2\kappa_2} = \sigma_1\mathbb{T}$. By induction hypothesis we deduce that there exist Δ'_2, \mathcal{C}_2 , and σ_2 solution for \mathcal{C}_2 such that $\Gamma_2, x : t = \sigma_2\Delta'_2$. Then it must be the case that $\Delta'_2 = \Delta_2, x : S$ for some Δ_2 and S such that $\Gamma_2 = \sigma_2\Delta_2$ and $t = \sigma_2S$. Since all type variables chosen by the type reconstruction algorithm are fresh, we know that $\text{dom}(\sigma_1) \cap \text{dom}(\sigma_2) = \emptyset$. Take $\sigma' \stackrel{\text{def}}{=} \sigma_1 \cup \sigma_2 \cup \{\varrho_1 \mapsto \kappa_1, \varrho_2 \mapsto \kappa_2\}$. Observe that $\sigma'\Delta_1 + \sigma'\Delta_2 = \Gamma_1 + \Gamma_2$ which is defined. By Lemma B.4 we deduce that there exist Δ, \mathcal{C}_3 , and $\sigma \supseteq \sigma'$ such that $\Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \mathcal{C}_3$ and σ is a solution for \mathcal{C}_3 that covers Δ . Take $\mathcal{C} \stackrel{\text{def}}{=} \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\mathbb{T} \hat{=} [S]^{1+\varrho_1, 2\varrho_2}\}$. Then σ is a solution for \mathcal{C} , because $\sigma\mathbb{T} = [t]^{1+\kappa_1, 2\kappa_2} = [\sigma S]^{1+\sigma(\varrho_1), 2\sigma(\varrho_2)} = \sigma[S]^{1+\varrho_1, 2\varrho_2}$. Also, by Lemma B.1 we have $\sigma\Delta = \sigma\Delta_1 + \sigma\Delta_2 = \Gamma_1 + \Gamma_2 = \Gamma$. We conclude $P \blacktriangleright \Delta; \mathcal{C}$ with an application of [I-IN].

$\boxed{\text{[T-PAR]}}$ Then $P = P_1 \mid P_2$ and $\Gamma = \Gamma_1 + \Gamma_2$ and $\Gamma_i \vdash P_i$ for $i = 1, 2$. By induction hypothesis we deduce that, for every $i = 1, 2$, there exist Δ_i, \mathcal{C}_i , and σ_i solution for \mathcal{C}_i such that $P_i \blacktriangleright \Delta_i; \mathcal{C}_i$ and $\Gamma_i = \sigma_i \Delta_i$. We also know that $\text{dom}(\sigma_1) \cap \text{dom}(\sigma_2) = \emptyset$ because type/use variables are always chosen fresh. Take $\sigma' \stackrel{\text{def}}{=} \sigma_1 \cup \sigma_2$. By Lemma B.4 we deduce that there exist Δ, \mathcal{C}_3 , and $\sigma \supseteq \sigma'$ such that $\Delta_1 \sqcup \Delta_2 \rightsquigarrow \Delta; \mathcal{C}_3$ and σ is a solution for \mathcal{C}_3 that covers Δ . By Lemma B.1 we also deduce that $\sigma \Delta = \sigma \Delta_1 + \sigma \Delta_2 = \Gamma_1 + \Gamma_2 = \Gamma$. We conclude by taking $\mathcal{C} \stackrel{\text{def}}{=} \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3$ with an application of [I-PAR].

$\boxed{\text{[T-REP]}}$ Then $P = *Q$ and $\Gamma \vdash Q$ and $\text{un}(\Gamma)$. By induction hypothesis we deduce that there exist Δ', \mathcal{C}' , and σ' solution for \mathcal{C}' such that $Q \blacktriangleright \Delta'; \mathcal{C}'$ and $\Gamma = \sigma' \Delta'$. Obviously $\sigma' \Delta' + \sigma' \Delta'$ is defined, hence by Lemma B.4 we deduce that there exist Δ, \mathcal{C}'' , and $\sigma \supseteq \sigma'$ such that $\Delta' \sqcup \Delta' \rightsquigarrow \Delta; \mathcal{C}''$ and σ is a solution for \mathcal{C}'' . By Lemma B.1 we deduce $\sigma \Delta = \sigma \Delta' + \sigma \Delta' = \Gamma + \Gamma = \Gamma$, where the last equality follows from the hypothesis $\text{un}(\Gamma)$ and Definition 3.7. We conclude $P \blacktriangleright \Delta; \mathcal{C}$ with an application of [I-REP] by taking $\mathcal{C} \stackrel{\text{def}}{=} \mathcal{C}' \cup \mathcal{C}''$. \square

APPENDIX C. SUPPLEMENT TO SECTION 5

Below is the derivation showing the reconstruction algorithm at work on the process (5.6).

$$\frac{\frac{a : \alpha_1 \blacktriangleright a : \alpha_1; \emptyset \quad 3 : \text{int} \blacktriangleright \emptyset; \emptyset}{a!3 \blacktriangleright a : \alpha_1; \{\alpha_1 \hat{=} [\text{int}]^{2\varrho_1, 1+\varrho_2}\}} \text{[I-OUT]} \quad \frac{b : \beta \blacktriangleright b : \beta; \emptyset \quad a : \alpha_2 \blacktriangleright a : \alpha_2; \emptyset}{b!a \blacktriangleright a : \alpha_2, b : \beta; \{\beta \hat{=} [\alpha_2]^{2\varrho_3, 1+\varrho_4}\}} \text{[I-OUT]}}{\frac{a!3 \mid b!a \blacktriangleright a : \alpha, b : \beta; \{\alpha \hat{=} \alpha_1 + \alpha_2, \alpha_1 \hat{=} [\text{int}]^{2\varrho_1, 1+\varrho_2}, \beta \hat{=} [\alpha_2]^{2\varrho_3, 1+\varrho_4}\}}{\text{new } a \text{ in } (a!3 \mid b!a) \blacktriangleright b : \beta; \{\alpha \hat{=} [\delta]^{e_5, e_5}, \alpha \hat{=} \alpha_1 + \alpha_2, \dots\}} \text{[I-NEW]}} \text{[I-PAR]}$$

Below is the derivation showing the reconstruction algorithm at work on the process (5.7). Only the relevant differences with respect to the derivation above are shown.

$$\frac{\frac{\frac{\vdots \quad \vdots}{b!a \blacktriangleright a : \alpha_2, b : \beta; \{\beta \hat{=} [\alpha_2]^{2\varrho_3, 1+\varrho_4}\}} \quad \frac{\vdots \quad \vdots}{c!a \blacktriangleright a : \alpha_3, c : \gamma; \{\gamma \hat{=} [\alpha_3]^{2\varrho_5, 1+\varrho_6}\}}}{\vdots \quad b!a \mid c!a \blacktriangleright a : \alpha_{23}, b : \beta, c : \gamma; \{\alpha_{23} \hat{=} \alpha_2 + \alpha_3, \dots\}} \text{[I-PAR]}}{\frac{a!3 \mid b!a \mid c!a \blacktriangleright a : \alpha, b : \beta, c : \gamma; \{\alpha \hat{=} \alpha_1 + \alpha_{23}, \dots\}}{\text{new } a \text{ in } (a!3 \mid b!a \mid c!a) \blacktriangleright b : \beta, c : \gamma; \{\alpha \hat{=} [\delta]^{e_5, e_5}, \alpha \hat{=} \alpha_1 + \alpha_{23}, \dots\}} \text{[I-NEW]}} \text{[I-PAR]}$$

Below is the derivation showing the reconstruction algorithm at work on the process (5.8).

$$\frac{\frac{b : \beta \blacktriangleright b : \beta; \emptyset \quad x : \gamma \blacktriangleright x : \gamma; \emptyset}{b!x \blacktriangleright b : \beta, x : \gamma; \{\beta \hat{=} [\gamma]^{2\varrho_1, 1+\varrho_2}\}} \text{[I-OUT]}}{a?(x).b!x \blacktriangleright a : \alpha, b : \beta; \{\alpha \hat{=} [\gamma]^{1+\varrho_3, 2\varrho_4}, \beta \hat{=} [\gamma]^{2\varrho_1, 1+\varrho_2}\}} \text{[I-IN]}$$