IRIS AperTO

UNIVERSITÀ DEGLI STUDI DI TORINO

AperTO - Archivio Istituzionale Open Access dell'Università di Torino

Type Reconstruction for the Linear π-Calculus with Composite and Equi-Recursive Types

This is a pre print version of the following article:

Original Citation:

Availability:

This version is available http://hdl.handle.net/2318/137787 since

Terms of use:

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

17 November 2026

Type Reconstruction for the Linear π-Calculus with Composite and Equi-Recursive Types

Luca Padovani – Dipartimento di Informatica, Università di Torino, Italy

Abstract. We generalize the linear π-calculus so that multiple processes can access the same composite, possibly recursive data structure containing linear values, provided that there is no overlapping access on such values. We define a complete type reconstruction algorithm for the type system. *En passant*, we solve the problem of reconstruction for equi-recursive session types.

1 Introduction

The linear π-calculus [9] is a formal model of communicating processes where channels are either *unlimited* or *linear*. Unlimited channels can be used for any number of input/output operations, while linear channels can only be used once for an input and once for an output. The ephemeral nature of linear channels contrasts with their value. In fact, they occur frequently in actual systems, they admit efficient implementations and enable important optimizations [6,5,9], and communications on linear channels are interference-free and partially confluent [9,10]. In addition, linearized channels can be encoded using linear channels [8,2] meaning that the vast body of research done on session types initiated with [3,4] has its foundations just on the linear π-calculus.

Type reconstruction is the problem of *inferring* the type of entities – channels in our case – given a program using them. For the linear π-calculus this problem was addressed in [7], although that work did not consider composite or recursive types. The goal of this work is the definition of a type reconstruction algorithm for the linear π-calculus extended with pairs, disjoint sums, and equi-recursive types. These constructs, 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.

By definition, linear channels can only be used for one shot communications, but it is a known fact that more sophisticated interactions can be implemented taking advantage of *channel mobility* using a continuation-passing style [8,2]. The basic idea is that, along with the proper payload of a communication, one can send another channel on which the rest of the converation takes place. To illustrate this mechanism, consider the process definitions

$$P(x,y) \stackrel{\text{def}}{=} (\nu a)(x!\langle y,a\rangle \mid P\langle a,y+1\rangle) \qquad C(x) \stackrel{\text{def}}{=} x?(y,z).C\langle z\rangle \tag{1.1}$$

modeling a *producer* P that sends messages to a *consumer* C. At each iteration, the producer creates a new channel a, sends the payload y to the consumer on x along with the continuation a on which subsequent communications will take place, and iterates. In parallel, the consumer waits for the payload and the continuation from the producer on x and then iterates. Explicit continuations are key to preserve the order of produced

messages. Had we modeled producer and consumer using the same x channel at each iteration, there would be no guarantee that messages were received in order.

Let us now assign types to the channels in (1.1) starting from x in the consumer. There, x is used once for receiving a pair made of an integer y and another channel z. Say the type of z is t and note that z is used in $C\langle z\rangle$ in the same place as x, meaning that x and z must have the same type. Then, also x has type t and t must satisfy the equation $t = [\mathtt{int} \times t]^{1,0}$ where the numbers 1 and 0, henceforth called *uses*, indicate that the channel of type t is used once for input and never for output. Channel x is used once in the producer for sending an integer y and a continuation a. Clearly, the continuation must have type t for that is how it is used in C. Therefore, x in P has type $s = [\mathtt{int} \times t]^{0,1}$ where the uses 0 and 1 indicate that x never used for input and is used once for output. Finally, there are two (non-binding) occurrences of a in the producer: a in $x!\langle n, a\rangle$ has type t because that is how a will be used by the consumer; a in $P\langle a, y+1\rangle$ has type s because that is how a will be used by the producer. Overall, the uses in the types of a say that a is a linear channel: it is used once by the producer for sending the payload and once by the consumer for receiving it.

We note three facts. First, the explicit continuation-passing style easily leads to channels with recursive types. This recursion represents an iterative communication behavior that takes place over channels rather than nesting structure of some data. Second, linear channels, like a, may syntactically occur multiple times and in general are used twice, once for input and once for output. Third, different occurrences of the same channel may have different yet *compatible* types. In the case of (1.1), the types t and s of the non-binding occurrences of a are compatible because corresponding uses in t and s are never 1 at the same time. The binding occurrence of a in (νa) has type $[\mathtt{int} \times t]^{1,1}$, which is the *composition* of t and s.

One legitimate question is whether and how the notions of type compatibility and combination extend from channel to composite types. In this respect, the existing bibliography provides rather unsatisfactory answers: the standard references on (type reconstruction for) the linear π-calculus [9,7] do not consider composite or recursive types. Channels in these works are polyadic, meaning that there is an implicit notion of tuple type, but there are no actual values of this type: tuples are created just before and destroyed right after each communication. Linear type systems with composite types have been discussed in [5,6] for the linear π-calculus and in [11] for a functional language. All these works, however, see linearity as a "contagious" property: every structure that contains linear values becomes linear itself. Such interpretation may be appropriate in a sequential setting, but is not the only sensible one in a concurrent/parallel setting. In fact, it is acceptable (and desirable, for the sake of parallelism) that several processes simultaneously access the *same* composite data structure, provided that they access *different* linear values stored therein. The problem is whether the type system is expressive enough to capture the fact that there are no overlapping accesses to the same linear values. For example, consider the type satisfying the equation

$$t_{list} = \mathtt{unit} \oplus ([\mathtt{int}]^{0,1} \times t_{list})$$

which is the disjoint sum between $\mathtt{unit}$ and the product $[\mathtt{int}]^{0,1} \times t_{list}$ and which can be used for describing lists of linear channels with type $[\mathtt{int}]^{0,1}$. If we follow [11,5,6],

an identifier l having type t_{list} can syntactically occur only once in a program, and a composition like for instance $Odd\langle l\rangle \mid Even\langle l\rangle$ is illegal. However, suppose that *Odd* and *Even* are the two processes defined by

$$\begin{array}{ll} Odd(x) \stackrel{\text{def}}{=} \texttt{case}\ x\ \texttt{of}\ [\,] \Rightarrow \mathbf{0} & \qquad Even(x) \stackrel{\text{def}}{=} \texttt{case}\ x\ \texttt{of}\ [\,] \Rightarrow \mathbf{0} \\ \qquad\qquad\quad y : z \Rightarrow y!\langle 3\rangle \mid Even\langle z\rangle & \qquad\qquad\quad y : z \Rightarrow Odd\langle z\rangle \end{array}$$

which walk through a list x: if x is the empty list `[]` they do nothing; if x has head y and tail z, *Odd* sends 3 on y and continues as $Even\langle z\rangle$ while *Even* ignores y and continues as $Odd\langle z\rangle$. So, $Odd\langle l\rangle$ sends 3 on every odd-indexed channel in l and $Even\langle l\rangle$ sends 3 on every even-indexed channel in l. The fact that *Odd* and *Even* access different linear values of a list is reflected in the two (mutually recursive) types of x in *Odd* and *Even*:

$$t_{odd} = \texttt{unit} \oplus ([\texttt{int}]^{0,1} \times t_{even}) \qquad \text{and} \qquad t_{even} = \texttt{unit} \oplus ([\texttt{int}]^{0,0} \times t_{odd}) \tag{1.2}$$

where the two 0's in t_{even} denote that *Even* does not use at all the first (and more generally every odd-indexed) element of its parameter x. The key observation is that just like a was allowed to occur twice in (1.1) with two compatible types t and s whose composition was $[\texttt{int} \times t]^{1,1}$, then we can allow l to occur twice in $Odd\langle l\rangle \mid Even\langle l\rangle$ given that the two occurrences of l are used according to two compatible types t_{odd} and t_{even} whose composition is t_{list}. The "only" difference is that, while t and s were channel types and their composition could be expressed simply by composing the uses in them, t_{odd} and t_{even} are recursive, structured types that compose to t_{list} *in the limit*.

To summarize the contributions of this paper:

- We study a natural generalization of the linear π-calculus that allows multiple processes to simultaneously access composite data structures containing linear values.
- We define a complete reconstruction algorithm for the type system which supports basic and channel types, pairs, disjoint sums, and equi-recursion.
- Given that sessions can be fully encoded into the linear π-calculus [8,2], we indirectly provide a complete reconstruction algorithm for equi-recursive session types.

We proceed with the formal definition of the language and of the type system (Section 2). The type reconstruction algorithm consists of a constraint generation phase (Section 3) and a constraint solving phase (Section 4). Section 5 concludes. The full version of the paper (with proofs) and a Haskell implementation of the algorithm are available at `http://www.di.unito.it/~padovani/hypha/`.

2 The Linear π-Calculus

We use the following conventions: we let m, n, ... range over natural numbers; we use a countable set of *channels* a, b, ... and a disjoint countable set of *variables* x, y, ...; *names* u, v, ... are either channels or variables. We work with the asynchronous π-calculus extended with constants, pairs, and disjoint sums. The syntax of expressions and processes is defined below:

$$\begin{array}{rcl} e & ::= & n \mid u \mid e,e \mid \texttt{inl}\ e \mid \texttt{inr}\ e \mid \cdots \\ P & ::= & \mathbf{0} \mid u?(x).P \mid u!\langle e\rangle \mid (P \mid Q) \mid *P \mid (\nu a)P \mid \begin{array}{l}\texttt{let}\ x,y = e \\ \texttt{in}\ P\end{array} \mid \begin{array}{l}\texttt{case}\ e\ \texttt{of} \\ \{\ \texttt{inl}\ x \Rightarrow P, \\ \quad \texttt{inr}\ y \Rightarrow Q\ \}\end{array} \end{array}$$

Table 1. Reduction of processes.

$$[\text{R-COMM}]\quad a!\langle \mathsf{v}\rangle \mid a?(x).Q \xrightarrow{a} Q\{\mathsf{v}/x\} \qquad [\text{R-LET}]\quad \mathtt{let}\ x,y = \mathsf{v},\mathsf{w}\ \mathtt{in}\ P \xrightarrow{\tau} P\{\mathsf{v},\mathsf{w}/x,y\}$$

$$[\text{R-CASE}]\quad \frac{k \in \{\mathtt{inl},\mathtt{inr}\}}{\mathtt{case}\ (k\ \mathsf{v})\ \mathtt{of}\{i\ x_i \Rightarrow P_i\}_{i=\mathtt{inl},\mathtt{inr}} \xrightarrow{\tau} P_k\{\mathsf{v}/x_k\}} \qquad [\text{R-PAR}]\quad \frac{P \xrightarrow{\ell} P'}{P \mid Q \xrightarrow{\ell} P' \mid Q}$$

$$[\text{R-NEW 1}]\quad \frac{P \xrightarrow{a} Q}{(\nu a)P \xrightarrow{\tau} (\nu a)Q} \qquad [\text{R-NEW 2}]\quad \frac{P \xrightarrow{\ell} Q \qquad \ell \neq a}{(\nu a)P \xrightarrow{\ell} (\nu a)Q}$$

$$[\text{R-STRUCT}]\quad \frac{P \equiv P' \qquad P' \xrightarrow{\ell} Q' \qquad Q' \equiv Q}{P \xrightarrow{\ell} Q}$$

Expressions $e, \ldots$ are either integers, names, pairs e_1, e_2 of expressions, or the injection $(i\ e)$ of an expression e using the constructor $i \in \{\mathtt{inl},\mathtt{inr}\}$. Expressions can be easily extended with additional data types and constructors without invalidating the results that follow. *Values* $\mathsf{v}, \mathsf{w}, \ldots$ are expressions without variables.

Processes $P, Q, \ldots$ comprise the standard constructs of the asynchronous π-calculus: the idle process $\mathbf{0}$ performs no actions; the input process $u?(x).P$ waits for a message from channel u, binds the message to x, and continues as P; the output process $u!\langle e\rangle$ sends (the value of) e on channel u; the parallel composition $P \mid Q$ executes P and Q in parallel; the replication $*P$ denotes infinitely many copies of P executing in parallel; the restriction $(\nu a)P$ creates a new channel a with scope P. In addition to these, we include two process forms for deconstructing pairs and disjoint sums. In particular, the process $\mathtt{let}\ x_1, x_2 = e\ \mathtt{in}\ P$ evaluates e, which must result into a pair $\mathsf{v}_1, \mathsf{v}_2$, binds each v_i to x_i, and continues as P. The process $\mathtt{case}\ e\ \mathtt{of}\{i\ x_i \Rightarrow P_i\}_{i=\mathtt{inl},\mathtt{inr}}$ evaluates e, which must result into a value $(i\ \mathsf{v})$ for $i \in \{\mathtt{inl},\mathtt{inr}\}$, binds v to x_i and continues as P_i.

Notions of *free names* $\mathsf{fn}(P)$ and *bound names* $\mathsf{bn}(P)$ of P are as expected. We identify processes modulo renaming of bound names and we write $P\{\mathsf{v}/x\}$ for the capture-avoiding substitution of the free occurrences of x in P with v.

The operational semantics of the language is defined in terms of a structual congruence relation, which rearranges terms that we do not want to distinguish, and a reduction relation, as usual. *Structural congruence* $\equiv$ is completely standard (in particular, it includes the law $*P \equiv *P \mid P$). *Reduction* is defined in Table 1 and is also conventional, except that we decorate the reduction relation with *labels* ℓ that are either channels or the special symbol τ, denoting an unobservable action: in [R-COMM] the label is identified as the channel a on which a message is exchanged, while in [R-LET] and [R-CASE] it is τ to denote the fact that these are internal computations not involving communications. 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 crosses the binder for a. Rule [R-STRUCT] closes reduction under structural congruence.

The type system makes use of a countable set of *type variables* $\alpha, \beta, \ldots$ and of *uses* $\kappa, \ldots$ which are elements of the set $\{0, 1, \omega\}$. Types $t, s, \ldots$ are defined by

$$t ::= \mathtt{int} \mid \alpha \mid t \times t \mid t \oplus t \mid [t]^{\kappa,\kappa} \mid \mu\alpha.t$$

and include the type of integers `int`, products $t_1 \times t_2$ denoting values of the form $(\mathsf{v}_1, \mathsf{v}_2)$ where v_i has type t_i for $i = 1,2$, and disjoint sums $t_1 \oplus t_2$ denoting values of the form $(\texttt{inl}\ \mathsf{v})$ when v has type t_1 or of the form $(\texttt{inr}\ \mathsf{v})$ when v has type t_2. Throughout the paper we let $\odot$ stand for either $\times$ or $\oplus$. 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: 0 for no use, 1 for a single use, and ω for any number of uses. For example: a channel with type $[t]^{1,1}$ must be used once for receiving a message of type t *and* once for sending a message of type t; a channel with type $[t]^{0,0}$ cannot be used; a channel with type $[t]^{\omega,\omega}$ can be used any number of times for sending and/or receiving. Type variables and recursions are used for building recursive types, as usual. Notion of free and bound type variables are as expected. We assume that every bound type variable is guarded by a constructor to avoid meaningless terms such as $\mu\alpha.\alpha$. We identify types modulo renaming of bound type variables and if their infinite unfoldings are the same (regular) tree [1]. In particular, we have $\mu\alpha.t = t\{\mu\alpha.t/\alpha\}$ where $t\{s/\alpha\}$ is the capture-avoiding substitution of the free occurrences of α in t with s.

We now define some key relations on uses and types. In particular, $\leq$ is the least *partial order* such that $0 \leq \kappa$ and *compatibility* $\asymp$ is the least relation such that

$$0 \asymp \kappa \qquad \kappa \asymp 0 \qquad \omega \asymp \omega \tag{2.1}$$

The meaning of $\leq$ is the obvious one: when $\kappa_1 \leq \kappa_2$, the use κ_2 allows at least as many operations on a channel as the use κ_1. In what follow we will write $\kappa_1 < \kappa_2$ if $\kappa_1 \leq \kappa_2$ and $\kappa_1 \neq \kappa_2$ and $\kappa_1 \vee \kappa_2$ for the *least upper bound* of κ_1 and κ_2, when it is defined. Compatibility determines whether the least upper bound of two uses expresses their combination without any loss of precision. For example, $0 \asymp 1$ because the combination of 0 uses and 1 use is $0 \vee 1 = 1$ use. On the contrary, $1 \not\asymp 1$ because there is no 2 use that expresses the combination of 1 and 1 and ω is less precise than 2. Similarly, $1 \not\asymp \omega$ because there is no use expressing the fact that a channel is used *at least* once.

Every binary relation $\mathscr{R}_{use}$ on uses induces a corresponding relation $\mathscr{R}_{type}$ on types, defined coinductively by the following rules:

$$\texttt{int}\ \mathscr{R}_{type}\ \texttt{int} \qquad \frac{\kappa_1\ \mathscr{R}_{use}\ \kappa_3 \qquad \kappa_2\ \mathscr{R}_{use}\ \kappa_4}{[t]^{\kappa_1,\kappa_2}\ \mathscr{R}_{type}\ [t]^{\kappa_3,\kappa_4}} \qquad \frac{t_1\ \mathscr{R}_{type}\ s_1 \qquad t_2\ \mathscr{R}_{type}\ s_2}{t_1 \odot t_2\ \mathscr{R}_{type}\ s_1 \odot s_2} \tag{2.2}$$

Similarly, the partial operation $\vee$ on uses coinducively induces one on types so that $t \vee s$ is the least upper bound of t and s, if it is defined. Note that $\leq$ is antisymmetric, in particular $t = s$ if and only if $t \leq s$ and $t \geq s$. Note also that $[t]^{\kappa_1,\kappa_2}\ \mathscr{R}\ [s]^{\kappa_3,\kappa_4}$ holds provided that $t = s$. The relation $t \asymp t$ is particularly interesting, because it gives us a neat characterization of unlimited types, those denoting values that must not or need not be used. Linear types, on the other hand, denote values that must be used:

Definition 2.1. *We say that t is* unlimited *if $t \asymp t$. We say that t is* linear *otherwise.*

Channel types are either limited or unlimited depending on their uses. So, $[t]^{1,0}$, $[t]^{0,1}$, and $[t]^{1,1}$ are all linear types whereas $[t]^{0,0}$ and $[t]^{\omega,\omega}$ are both unlimited. Other types are linear or unlimited according to the channel types occurring in them. For

Table 2. Type rules for expressions and processes.

Expressions

$$\text{[T-CONST]}\ \frac{\Gamma \asymp \Gamma}{\Gamma \vdash n : \mathtt{int}} \qquad \text{[T-NAME]}\ \frac{\Gamma \asymp \Gamma}{\Gamma, u : t \vdash u : t} \qquad \text{[T-PAIR]}\ \frac{\Gamma \vdash e : t \qquad \Gamma' \vdash e' : s}{\Gamma + \Gamma' \vdash e, e' : t \times s}$$

$$\text{[T-INL]}\ \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathtt{inl}\ e : t \oplus s} \qquad \text{[T-INR]}\ \frac{\Gamma \vdash e : s}{\Gamma \vdash \mathtt{inr}\ e : t \oplus s}$$

Processes

$$\text{[T-IDLE]}\ \frac{\Gamma \asymp \Gamma}{\Gamma \vdash \mathbf{0}} \qquad \text{[T-IN]}\ \frac{\Gamma, x : t \vdash P \qquad 0 < \kappa}{\Gamma + u : [t]^{\kappa,0} \vdash u?(x).P} \qquad \text{[T-OUT]}\ \frac{\Gamma \vdash e : t \qquad 0 < \kappa}{\Gamma + u : [t]^{0,\kappa} \vdash u!\langle e \rangle}$$

$$\text{[T-PAR]}\ \frac{\Gamma_i \vdash P_i \ ^{(i=1,2)}}{\Gamma_1 + \Gamma_2 \vdash P_1 \mid P_2} \qquad \text{[T-REP]}\ \frac{\Gamma \vdash P}{\Gamma + \Gamma \vdash *P}$$

$$\text{[T-NEW]}\ \frac{\Gamma, a : [t]^{\kappa,\kappa} \vdash P}{\Gamma \vdash (\nu a)P} \qquad \text{[T-LET]}\ \frac{\Gamma \vdash e : t \times s \qquad \Gamma', x : t, y : s \vdash P}{\Gamma + \Gamma' \vdash \mathtt{let}\ x, y = e\ \mathtt{in}\ P}$$

$$\text{[T-CASE]}\ \frac{\Gamma \vdash e : t \oplus s \qquad \Gamma', x_i : t \vdash P_i \ ^{(i=\mathtt{inl},\mathtt{inr})}}{\Gamma + \Gamma' \vdash \mathtt{case}\ e\ \mathtt{of}\ \{i\ x_i \Rightarrow P_i\}_{i=\mathtt{inl},\mathtt{inr}}}$$

example, $[t]^{0,0} \times [t]^{1,0}$ is linear while $[t]^{0,0} \times [t]^{0,\omega}$ is unlimited. Recursion does not affect the classification of types into linear and unlimited. For example, $\mu\alpha.[\mathtt{int} \times \alpha]^{1,0}$ is a linear type that denotes a channel that must be used once for receiving a pair made of an integer and another channel with the same type.

We type check expressions and processes in *type environments* Γ, ..., which are finite maps from names to types that we write as $u_1 : t_1, \ldots, u_n : t_n$. We identify type environments modulo the order of their bindings, we denote the *empty environment* with $\emptyset$, we write $\mathsf{dom}(\Gamma)$ for the *domain* of Γ, namely the set of names for which there is a binding in Γ, and Γ_1, Γ_2 for the *union* of Γ_1 and Γ_2 when $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \emptyset$. We extend the relation $\asymp$ between types pointwise to type environments. Following [9] we define a partial operation $+$ to combine type environments:

$$\emptyset + \Gamma = \Gamma + \emptyset \stackrel{\text{def}}{=} \Gamma$$
$$(\Gamma_1, u : t) + (\Gamma_2, u : s) \stackrel{\text{def}}{=} (\Gamma_1 + \Gamma_2), u : t \vee s \qquad \text{if } t \asymp s$$

Note that $\Gamma_1 + \Gamma_2$ is undefined if there is $u \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)$ and $\Gamma_1(u) \not\asymp \Gamma_2(u)$ and that $\mathsf{dom}(\Gamma_1 + \Gamma_2) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$. 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 must be used according to compatible types, and its type in $\Gamma_1 + \Gamma_2$ is their least upper bound. For example, if a channel a is used by a process for sending a message of type $\mathtt{int}$, it has type $[\mathtt{int}]^{0,1}$ in that process; if it is used by anther process for receiving a message of type $\mathtt{int}$, it has type $[\mathtt{int}]^{1,0}$ in that process. Overall, the channel is used by the two processes for sending and receiving a message of type $\mathtt{int}$, namely according to the type $[\mathtt{int}]^{0,1} \vee [\mathtt{int}]^{1,0} = [\mathtt{int}]^{1,1}$.

Type rules for expressions and processes are presented in Table 2. In general these rules are basically the same as those found in the literature [9,7]. All the additional

flexibility enabled by our type discipline is actually a consequence of the more general notion of type combination $\vee$. The rules for expressions are unremarkable. Just observe that the part of the type environment that is not used in the expression must be unlimited ($\Gamma \asymp \Gamma$), and that [T-PAIR] uses $+$ for combining the type environments that are necessary for typing the two components of the pair. This makes it possible to type an expression such as (a,a), for example in the type environment $a : [\texttt{int}]^{1,1}$. The idle process does nothing, so it is well typed only in an unlimited environment. Input and output processes require a strictly positive use of the corresponding operation in the type of the channel u used for communication. Rule [T-REP] states that a replicated process $*P$ is well typed in the environment $\Gamma + \Gamma$ provided that P is well typed in Γ. The rationale for this is that $*P$ stands for an unbounded number of copies of P composed in parallel, hence each copy of P requires a corresponding copy Γ of the type environment. In principle this would require typing $*P$ in the environment $\Gamma + \Gamma + \cdots$, but because of the way $+$ is defined, we know that $\Gamma + \Gamma = \Gamma$ whenever $\Gamma + \Gamma$ is defined. The rules [T-PAR], [T-LET], and [T-CASE] are conventional. Note again the use of $+$ for combining environments used in different parts of the process. Finally, rule [T-NEW] states that $(\nu a)P$ is well typed if so is P, where P has visibility of a. We require the type of a to have the same uses for input and output. This is not necessary for the soundness of the type system, although it is a reasonable choice in practice.

The type system is sound and the results in Section 4.3 of [9] can be formulated in our setting. In particular, the operations on a channel never exceed the uses in its type. It is possible to establish other basic safety properties, for instance that closed, well-typed `let`'s and `case`'s always reduce. The long version of the paper gives more details.

Example 2.1. We model a recursive process definition using unlimited channels: a replicated input on the channel represents the definition, while an output on the channel represents an invocation of the definition. For example, for *Odd* and *Even* we have

$$
\begin{array}{l}
*a?(x).\texttt{case}\ x\ \texttt{of} \\
\quad \texttt{inl}\ y_1 \Rightarrow \mathbf{0} \\
\quad \texttt{inr}\ y_2 \Rightarrow \texttt{let}\ y,z = y_2\ \texttt{in}\ y!\langle 3\rangle \mid b!\langle z\rangle
\end{array}
\qquad\qquad
\begin{array}{l}
*b?(x).\texttt{case}\ x\ \texttt{of} \\
\quad \texttt{inl}\ y_1 \Rightarrow \mathbf{0} \\
\quad \texttt{inr}\ y_2 \Rightarrow \texttt{let}\ y,z = y_2\ \texttt{in}\ a!\langle z\rangle
\end{array}
$$

and we assume that `inl` 0 represents the empty list `[]` and `inr` (y,z) represents the non-empty list y `:` z with head y and tail z. Below is a derivation showing that *Odd* encoded as shown above is well typed, where we take t_{odd} and t_{even} defined in (1.2):

$$
\dfrac{
\dfrac{}{y_1 : \texttt{int} \vdash \mathbf{0}}\,[\text{T-IDLE}]
\qquad
\dfrac{
\dfrac{}{y : [\texttt{int}]^{0,1} \vdash y!\langle 3\rangle}\,[\text{T-OUT}]
\qquad
\dfrac{}{b : [t_{even}]^{0,\omega}, z : t_{even} \vdash b!\langle z\rangle}\,[\text{T-OUT}]
}{
\dfrac{b : [t_{even}]^{0,\omega}, y : [\texttt{int}]^{0,1}, z : t_{even} \vdash y!\langle 3\rangle \mid b!\langle z\rangle}{b : [t_{even}]^{0,\omega}, y_2 : [\texttt{int}]^{0,1} \times t_{even} \vdash \texttt{let}\ y,z = y_2\ \texttt{in}\ \cdots}\,[\text{T-LET}]
}\,[\text{T-PAR}]
}{
\dfrac{
\dfrac{
\dfrac{b : [t_{even}]^{0,\omega}, x : t_{odd} \vdash \texttt{case}\ x\ \texttt{of}\ \cdots}{a : [t_{odd}]^{\omega,0}, b : [t_{even}]^{0,\omega} \vdash a?(x).\texttt{case}\ x\ \texttt{of}\ \cdots}\,[\text{T-IN}]
}{a : [t_{odd}]^{\omega,0}, b : [t_{even}]^{0,\omega} \vdash *a?(x).\texttt{case}\ x\ \texttt{of}\ \cdots}\,[\text{T-REP}]
}\,[\text{T-CASE}]
$$

Note that a and b must be unlimited channels because they occur free in a replicated process, for which rule [T-REP] requires an unlimited environment. A similar deriva-

tion shows that *Even* is well typed in an environment where the types of a and b have swapped uses

$$a : [t_{odd}]^{0,\omega}, b : [t_{even}]^{\omega,0} \vdash *b?(x).\texttt{case}\ x\ \texttt{of}\ \cdots$$

so the combined types of a and b are $[t_{odd}]^{\omega,\omega}$ and $[t_{even}]^{\omega,\omega}$ respectively. We conclude

$$a : [t_{odd}]^{\omega,\omega}, b : [t_{even}]^{\omega,\omega}, l : t_{list} \vdash a!\langle l\rangle \mid b!\langle l\rangle$$

because $t_{odd} \asymp t_{even}$ and $t_{odd} \vee t_{even} = t_{list}$. ■

3 Constraint Generation

The problem of type reconstruction is the following: given a process P, find a type environment Γ such that $\Gamma \vdash P$, provided there is one. In general we also want to maximize the number of channels that have a linear type. The rules shown in Table 2 rely on a fair amount of guesses regarding the structure of types in the type environment, how they are split/combined using $\vee$, and the uses occurring in them. So, these rules cannot be easily turned into a type reconstruction algorithm. The way we follow to define one is rather conventional: we give an alternative set of syntax-directed rules that compute *constraints* on types and uses and then we search for a solution of such constraints. The novelty is that we will need constraints expressing not only the *equality* between types and uses, but also the order $\leq$ and compatibility $\asymp$, which affect the solution phase in non-trivial ways.

To get started, we generalize uses to *use expressions*, which are either uses or *use variables* $\rho, \ldots$ that denote an unknown use. We also define *type expressions* as types without μ's and where we admit use expressions wherever uses can occur. We keep κ and t for ranging over use and type expressions and we say that t is *proper* if it is different from a type variable. *Constraints* $\varphi, \ldots$ have one of these forms:

$$\varphi \quad ::= \quad \kappa_1 \mathrel{\mathscr{R}_{\mathsf{c}}} \kappa_2 \quad \mid \quad t_1 \mathrel{\mathscr{R}_{\mathsf{c}}} t_2$$

where $\mathscr{R} \in \{\leq, <, \asymp, \sim\}$ and $\sim$ is the trivial relation between uses. The subscript $\cdot_{\mathsf{c}}$ reminds us that $\kappa_1 \mathrel{\mathscr{R}_{\mathsf{c}}} \kappa_2$ and $t_1 \mathrel{\mathscr{R}_{\mathsf{c}}} t_2$ are just *triples* made of two use or type expressions and a *symbol* $\mathscr{R}_{\mathsf{c}}$ denoting a relation. Equality constraints $=_{\mathsf{c}}$ can be expressed as the conjunction of two constraints $\leq_{\mathsf{c}}$ and $\geq_{\mathsf{c}}$, given that $\leq$ is antisymmetric. Constraints with the strict order $<_{\mathsf{c}}$ will be generated only for use expressions. Compatibility constraints $\asymp_{\mathsf{c}}$ will be generated for expressing type and use composition, as well as for expressing the assumption that a type/use is unlimited (see *e.g.* the premises of [T-IDLE]). Finally, $\sim_{\mathsf{c}}$ constraints propagate information on the structure of types. We will see in Section 4 how this information is used for verifying the satisfiability of constraints.

We let $\mathscr{C}, \ldots$ range over finite sets of constraints. The *domain* of $\mathscr{C}$, written $\mathsf{dom}(\mathscr{C})$, is the (finite) set of use and type expressions occurring in the constraints in $\mathscr{C}$. We let σ range over finite maps from type variables to types and from use variables to uses. The *application* of σ replaces use variables ρ and type variables α with the corresponding uses $\sigma(\rho)$ and types $\sigma(\alpha)$. We write $\sigma\kappa$ and σt for the application of σ to κ and t. We say that σ is a *solution* of $\mathscr{C}$ if $\sigma t \mathrel{\mathscr{R}} \sigma s$ for every $t \mathrel{\mathscr{R}_{\mathsf{c}}} s \in \mathscr{C}$ and $\sigma\kappa_1 \mathrel{\mathscr{R}} \sigma\kappa_2$ for every

$\kappa_1 \mathrel{\mathscr{R}_c} \kappa_2 \in \mathscr{C}$. We say that $\mathscr{C}$ is *satisfiable* if it has a solution. We say that $\mathscr{C}_1$ and $\mathscr{C}_2$ are *equivalent* if they have the same solutions.

We need two operators for combining and merging type environments in the reconstruction algorithm. They take two environments Γ_1 and Γ_2 and produce a pair consisting of another type environment Γ and a set of constraints $\mathscr{C}$:

$$\frac{\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \emptyset}{\Gamma_1 \sqcup \Gamma_2 \rightsquigarrow \Gamma_1, \Gamma_2; \emptyset} \qquad \frac{\Gamma_1 \sqcup \Gamma_2 \rightsquigarrow \Gamma; \mathscr{C} \qquad \alpha \text{ fresh}}{(\Gamma_1, u : t) \sqcup (\Gamma_2, u : s) \rightsquigarrow \Gamma, u : \alpha; \mathscr{C} \cup \{t \asymp_c s, t \leq_c \alpha, s \leq_c \alpha\}}$$

$$\emptyset \sqcap \emptyset \rightsquigarrow \emptyset; \emptyset \qquad \frac{\Gamma_1 \sqcap \Gamma_2 \rightsquigarrow \Gamma; \mathscr{C}}{(\Gamma_1, u : t) \sqcap (\Gamma_2, u : s) \rightsquigarrow \Gamma, u : t; \mathscr{C} \cup \{t =_c s\}}$$

The relation $\Gamma_1 \sqcup \Gamma_2 \rightsquigarrow \Gamma; \mathscr{C}$ combines the type environments Γ_1 and Γ_2 into Γ when the names in $\mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$ are used *both* as specified in Γ_1 *and also* in Γ_2, so $\sqcup$ is analogous to $+$ in Table 2. When Γ_1 and Γ_2 have disjoint domains, their combination is just their union and no constraints are generated. Any name u that occurs in $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)$ must be used according to compatible types $\Gamma_1(u) \asymp \Gamma_2(u)$ and its type must be an upper bound of both $\Gamma_1(u)$ and $\Gamma_2(u)$. In general $\Gamma_1(u)$ and $\Gamma_2(u)$ are type expressions with free type variables, hence these relations cannot be checked right away. Rather, they are symbolically recorded in the set of constraints $\mathscr{C}$. Note in particular that the combined type of u is unknown and is represented by a fresh type variable that is an upper bound of $\Gamma_1(u)$ and $\Gamma_2(u)$. The relation $\Gamma_1 \sqcap \Gamma_2 \rightsquigarrow \Gamma; \mathscr{C}$ merges the type environments Γ_1 and Γ_2 into Γ when the names in $\mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$ are used in alternative branches of a `case` construct. Note that $\Gamma_1 \sqcap \Gamma_2 \rightsquigarrow \Gamma; \mathscr{C}$ holds if and only if $\Gamma_1(u) = \Gamma_2(u)$ for every $u \in \mathsf{dom}(\Gamma_1) = \mathsf{dom}(\Gamma_2)$. This corresponds to the fact that in [T-CASE] we use the *same* type environment Γ' for typing the two branches of the `case`.

The rules to synthesize type environments and generate constraints are presented in Table 3 and derive judgments of the form $e : t \triangleright \Gamma; \mathscr{C}$ for expressions and $P \triangleright \Gamma; \mathscr{C}$ for processes. They closely correspond to those in Table 2; for this and space reasons we will not describe them in detail. In general, unknown uses and types become fresh use and type variables (all variables introduced by the rules are assumed to be fresh), every application of the $+$ operator in Table 2 becomes an application of $\sqcup$ in Table 3, and every assumption on uses and types becomes a constraint. Constraints accumulate from the premises to the conclusion of each rule. In rules [I-INL] and [I-INR] the type of the disjoint sum which was guessed in [T-INL] and [T-INR] is becomes a fresh type variable. In rules [I-IN] and [I-OUT] it is not known whether the used channel u is linear or unlimited, so the constraint $0 <_c \rho$ records the fact that ρ must be either 1 or ω. Rule [I-NEW] requires a to have a channel type with equal uses by having the same use variable ρ twice. There is also a rule [I-WEAK] that has no correspondence in Table 2. It is necessary because [I-IN], [I-NEW], [I-LET], and [I-CASE], which correspond to binding constructs of the calculus, *assume* that the bound names occur in the premises on these rules. This may not be the case if a bound name is never used. With rule [I-WEAK] we can introduce missing names in type environments wherever is convenient. Of course, an unused name must have a type α that is unlimited, which is recorded by the constraint $\alpha \asymp_c \alpha$. Strictly speaking, with [I-WEAK] this set of rules is not syntax directed, which in principle is a problem if we want to obtain an algorithm. In practice, the places where

Table 3. Constraint generation for expressions and processes.

Expressions

$$[\text{I-INT}]\quad n : \mathtt{int} \triangleright \emptyset;\emptyset \qquad [\text{I-NAME}]\quad u : \alpha \triangleright u : \alpha;\emptyset$$

$$[\text{I-PAIR}]\quad \frac{e_i : t_i \triangleright \Gamma_i;\mathscr{C}_i \ ^{(i=1,2)} \qquad \Gamma_1 \sqcup \Gamma_2 \rightsquigarrow \Gamma;\mathscr{C}_3}{e_1, e_2 : t_1 \times t_2 \triangleright \Gamma;\mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3} \qquad [\text{I-INL}]\quad \frac{e : t \triangleright \Gamma;\mathscr{C}}{\mathtt{inl}\ e : t \oplus \alpha \triangleright \Gamma;\mathscr{C}} \qquad [\text{I-INR}]\quad \frac{e : t \triangleright \Gamma;\mathscr{C}}{\mathtt{inr}\ e : \alpha \oplus t \triangleright \Gamma;\mathscr{C}}$$

Processes

$$[\text{I-IDLE}]\quad \mathbf{0} \triangleright \emptyset;\emptyset \qquad [\text{I-IN}]\quad \frac{P \triangleright \Gamma, x : t;\mathscr{C} \qquad \Gamma \sqcup u : [t]^{\rho,0} \rightsquigarrow \Gamma';\mathscr{C}'}{u?(x).P \triangleright \Gamma';\mathscr{C} \cup \mathscr{C}' \cup \{0 <_{\mathsf{c}} \rho\}} \qquad [\text{I-OUT}]\quad \frac{e : t \triangleright \Gamma;\mathscr{C} \qquad \Gamma \sqcup u : [t]^{0,\rho} \rightsquigarrow \Gamma';\mathscr{C}'}{u!\langle e\rangle \triangleright \Gamma';\mathscr{C}' \cup \{0 <_{\mathsf{c}} \rho\}}$$

$$[\text{I-PAR}]\quad \frac{P_i \triangleright \Gamma_i;\mathscr{C}_i \ ^{(i=1,2)} \qquad \Gamma_1 \sqcup \Gamma_2 \rightsquigarrow \Gamma;\mathscr{C}_3}{P_1 \mid P_2 \triangleright \Gamma;\mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3} \qquad [\text{I-REP}]\quad \frac{P \triangleright \Gamma;\mathscr{C} \qquad \Gamma \sqcup \Gamma \rightsquigarrow \Gamma';\mathscr{C}'}{*P \triangleright \Gamma';\mathscr{C} \cup \mathscr{C}'} \qquad [\text{I-WEAK}]\quad \frac{P \triangleright \Gamma;\mathscr{C}}{P \triangleright \Gamma, u : \alpha;\mathscr{C} \cup \{\alpha \asymp_{\mathsf{c}} \alpha\}}$$

$$[\text{I-NEW}]\quad \frac{P \triangleright \Gamma, a : t;\mathscr{C}}{(\nu a)P \triangleright \Gamma;\mathscr{C} \cup \{t =_{\mathsf{c}} [\alpha]^{\rho,\rho}\}} \qquad [\text{I-LET}]\quad \frac{e : t \triangleright \Gamma_1;\mathscr{C}_1 \qquad P \triangleright \Gamma_2, x : t_1, y : t_2;\mathscr{C}_2 \qquad \Gamma_1 \sqcup \Gamma_2 \rightsquigarrow \Gamma;\mathscr{C}_3}{\mathtt{let}\ x,y = e\ \mathtt{in}\ P \triangleright \Gamma;\mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \{t =_{\mathsf{c}} t_1 \times t_2\}}$$

$$[\text{I-CASE}]\quad \frac{e : t \triangleright \Gamma_1;\mathscr{C}_1 \qquad P_i \triangleright \Gamma_i, x_i : t_i;\mathscr{C}_i \ ^{(i=\mathtt{inl},\mathtt{inr})} \qquad \Gamma_{\mathtt{inl}} \sqcap \Gamma_{\mathtt{inr}} \rightsquigarrow \Gamma_2;\mathscr{C}_2 \qquad \Gamma_1 \sqcup \Gamma_2 \rightsquigarrow \Gamma_3;\mathscr{C}_3}{\mathtt{case}\ e\ \mathtt{of}\{i\ x_i \Rightarrow P_i\}_{i=\mathtt{inl},\mathtt{inr}} \triangleright \Gamma_3;\mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \mathscr{C}_{\mathtt{inl}} \cup \mathscr{C}_{\mathtt{inr}} \cup \{t =_{\mathsf{c}} t_{\mathtt{inl}} \oplus t_{\mathtt{inr}}\}}$$

[I-WEAK] may be necessary are easy to spot (in the premises of all the aforementioned rules for binding constructs). What we gain with [I-WEAK] is a simpler presentation of the rules for constraint generation.

There is a tight correspondence between the type system and constraint generation. Every satisfiable set of constraints generated from P corresponds to a typing for P.

Theorem 3.1. *If $P \triangleright \Gamma;\mathscr{C}$ and σ is a solution for $\mathscr{C}$, then $\sigma\Gamma \vdash P$.*

In fact, when $P \triangleright \Gamma;\mathscr{C}$ we can think of $\Gamma;\mathscr{C}$ as the *principal typing* of P, because any type environment Γ' such that $\Gamma' \vdash P$ can be obtained by applying a solution for $\mathscr{C}$ to Γ.

Theorem 3.2. *If $\Gamma' \vdash P$, then $P \triangleright \Gamma;\mathscr{C}$ for some Γ, $\mathscr{C}$ and σ solution of $\mathscr{C}$ and $\Gamma' = \sigma\Gamma$.*

Example 3.1. We show the constraint set generated by two processes accessing the same composite structure containing linear values. The *Odd* and *Even* processes in Section 1 are too large to be discussed in here, so we focus on a simpler, artificial process that exhibits the same phenomenon. We consider

$$a?(x).(\mathtt{let}\ y,z = x\ \mathtt{in}\ y!\langle 1\rangle \mid \mathtt{let}\ y,z = x\ \mathtt{in}\ z!\langle 2\rangle)$$

Table 4. Constraint solver algorithm.

Input: a finite set of constraints $\mathscr{C}$.
Output: either **fail** or a solution of $\mathscr{C}$.

1. Compute $\overline{\mathscr{C}}$;
2. Compute a solution σ_u for the use constraints in $\overline{\mathscr{C}}$, or **fail** if there is none;
3. If $t \sim_{\mathsf{c}} s \in \overline{\mathscr{C}}$ and t, s are proper and are different constructors, then **fail**;
4. Let $\sigma_t = \{\alpha \mapsto \mathsf{sup}_{\overline{\mathscr{C}},\sigma_u}(\{\alpha\}) \mid \alpha \in \mathsf{dom}(\overline{\mathscr{C}})\}$;
5. Return $\sigma_u \cup \sigma_t$.

which receives a pair x of channels from a and sends 1 and 2 on them. Note that the pair x is deconstructed twice, but every time only one of its components is used. We obtain

$$\dfrac{\dfrac{\dfrac{\dfrac{}{y!\langle 1\rangle \triangleright y : [\mathtt{int}]^{0,\rho_1}; \mathscr{C}_1}\,[\text{I-OUT}]}{y!\langle 1\rangle \triangleright y : [\mathtt{int}]^{0,\rho_1}, z : \gamma; \mathscr{C}_2}\,[\text{I-WEAK}]}{\mathtt{let}\ y,z = x\ \mathtt{in}\ y!\langle 1\rangle \triangleright x : \alpha_1; \mathscr{C}_3}\,[\text{I-LET}] \quad \dfrac{\dfrac{\dfrac{}{z!\langle 2\rangle \triangleright z : [\mathtt{int}]^{0,\rho_2}; \mathscr{C}_4}\,[\text{I-OUT}]}{z!\langle 2\rangle \triangleright y : \beta, z : [\mathtt{int}]^{0,\rho_2}; \mathscr{C}_5}\,[\text{I-WEAK}]}{\mathtt{let}\ y,z = x\ \mathtt{in}\ z!\langle 2\rangle \triangleright x : \alpha_2; \mathscr{C}_6}\,[\text{I-LET}]}{\mathtt{let}\ y,z = x\ \mathtt{in}\ y!\langle 1\rangle \mid \mathtt{let}\ y,z = x\ \mathtt{in}\ z!\langle 2\rangle \triangleright x : \alpha; \mathscr{C}_7}\,[\text{I-PAR}]$$
$$\dfrac{}{a?(x).(\mathtt{let}\ y,z = x\ \mathtt{in}\ y!\langle 1\rangle \mid \mathtt{let}\ y,z = x\ \mathtt{in}\ z!\langle 2\rangle) \triangleright a : [\alpha]^{\rho_3,0}; \mathscr{C}_8}\,[\text{I-IN}]$$

where

$$\mathscr{C}_1 \stackrel{\text{def}}{=} \{0 <_{\mathsf{c}} \rho_1\} \quad \mathscr{C}_2 \stackrel{\text{def}}{=} \mathscr{C}_1 \cup \{\gamma \asymp_{\mathsf{c}} \gamma\} \quad \mathscr{C}_3 \stackrel{\text{def}}{=} \mathscr{C}_2 \cup \{\alpha_1 =_{\mathsf{c}} [\mathtt{int}]^{0,\rho_1} \times \gamma\}$$
$$\mathscr{C}_4 \stackrel{\text{def}}{=} \{0 <_{\mathsf{c}} \rho_2\} \quad \mathscr{C}_5 \stackrel{\text{def}}{=} \mathscr{C}_4 \cup \{\beta \asymp_{\mathsf{c}} \beta\} \quad \mathscr{C}_6 \stackrel{\text{def}}{=} \mathscr{C}_5 \cup \{\alpha_2 =_{\mathsf{c}} \beta \times [\mathtt{int}]^{0,\rho_2}\}$$
$$\mathscr{C}_7 \stackrel{\text{def}}{=} \mathscr{C}_3 \cup \mathscr{C}_6 \cup \{\alpha_1 \asymp_{\mathsf{c}} \alpha_2, \alpha_1 \leq_{\mathsf{c}} \alpha, \alpha_2 \leq_{\mathsf{c}} \alpha\} \quad \mathscr{C}_8 \stackrel{\text{def}}{=} \mathscr{C}_7 \cup \{0 <_{\mathsf{c}} \rho_3\}$$

Within each `let` the variable x is assigned a distinct type variable α_i. Eventually, [I-PAR] finds out that x occurs twice, so it records in $\mathscr{C}_7$ the fact that the two types α_1 and α_2 must be compatible and that the overall type α of x must be an upper bound of both. ■

4 Constraint Solving

In this section we define an algorithm that, given a finite set of constraints $\mathscr{C}$, determines whether $\mathscr{C}$ is satisfiable and, if so, computes a solution of $\mathscr{C}$. The algorithm, sketched in Table 4, comprises 5 steps that can be roughly grouped in three phases: *saturation*, *verification*, and *synthesis*. The phases are detailed in the rest of the section.

Saturation (step 1). The $\leq_{\mathsf{c}}$ and $\asymp_{\mathsf{c}}$ constraints determined during constraint generation relate type expressions, but they are meant to affect the use variables occurring in these type expressions (recall from (2.2) that every relation $\mathscr{R}_{type}$ between types is the extension of $\mathscr{R}_{use}$ between uses). In order to find all constraints that must hold between

use expressions, the set $\mathscr{C}$ is *saturated* with all the constraints that are entailed by those already in $\mathscr{C}$. Entailment is expressed thorugh a binary relation $\vDash$ defined as follows:

$$
\begin{array}{lcl}
\text{[E-REFL]} & \{t\ \mathscr{R}_{\mathsf{c}}\ s\} \vDash \{t\ \mathscr{R}_{\mathsf{c}}\ t, s\ \mathscr{R}_{\mathsf{c}}\ s\} & \mathscr{R} \in \{\leq, \sim\} \\
\text{[E-SYMM]} & \{t\ \mathscr{R}_{\mathsf{c}}\ s\} \vDash \{s\ \mathscr{R}_{\mathsf{c}}\ t\} & \mathscr{R} \in \{=, \sim\} \\
\text{[E-TRANS]} & \{t_1\ \mathscr{R}_{\mathsf{c}}\ t_2, t_2\ \mathscr{R}_{\mathsf{c}}\ t_3\} \vDash \{t_1\ \mathscr{R}_{\mathsf{c}}\ t_3\} & \mathscr{R} \in \{\leq, \sim\} \\
\text{[E-COMP 1]} & \{t_1 =_{\mathsf{c}} t_2, t_2 \asymp_{\mathsf{c}} t_3\} \vDash \{t_1 \asymp_{\mathsf{c}} t_3\} & \\
\text{[E-COMP 2]} & \{t_1 \leq_{\mathsf{c}} t_2, t_2 \asymp_{\mathsf{c}} t_3\} \vDash \{t_1 \asymp_{\mathsf{c}} t_3\} & \\
\text{[E-OPER]} & \{t_1 \odot t_2\ \mathscr{R}_{\mathsf{c}}\ s_1 \odot s_2\} \vDash \{t_1\ \mathscr{R}_{\mathsf{c}}\ s_1, t_2\ \mathscr{R}_{\mathsf{c}}\ s_2\} & \\
\text{[E-CHANNEL]} & \{[t]^{\kappa_1,\kappa_2}\ \mathscr{R}_{\mathsf{c}}\ [s]^{\kappa_3,\kappa_4}\} \vDash \{t =_{\mathsf{c}} s, \kappa_1\ \mathscr{R}_{\mathsf{c}}\ \kappa_3, \kappa_2\ \mathscr{R}_{\mathsf{c}}\ \kappa_4\} & \\
\text{[E-STRUCT]} & \{t\ \mathscr{R}_{\mathsf{c}}\ s\} \vDash \{t \sim_{\mathsf{c}} s\} &
\end{array}
$$

The first three rules [E-REFL], [E-SYMM], and [E-TRANS] just compute the reflexive, symmetric, and transitive closures of those relations that enjoy such properties. Rule [E-COMP 1] propagates compatibility constraints between equivalent types, and [E-COMP 2] propagates compatibility constraints "downwards" from a type to a smaller one (indeed, it is the case that $\kappa_1 \leq \kappa_2 \asymp \kappa_3$ implies $\kappa_1 \asymp \kappa_3$). Rule [E-OPER] propagates constraints between composite to their components. Rule [E-CHANNEL] propagates constraints from type to use expressions and imposes the equality of message types for related channel types. Finally, rule [E-STRUCT] generates trivial $\sim_{\mathsf{c}}$ constraints between any pair of related type expressions. This is necessary to make sure that all message type equality constrains are generated by [E-CHANNEL], given that $\asymp$ is not transitive. We denote by $\overline{\mathscr{C}}$ the smallest set that includes $\mathscr{C}$ and that is closed by the rules [E-*] above. Observe that every $\mathscr{C}$ generated by the rules in Table 3 is finite, and that the entialment rules [E-*] do not change the domain of the set $\mathscr{C}$ being saturated. Therefore, $\overline{\mathscr{C}}$ is always finite and can be computed in finite time by a simple iterative algorithm that repeatedly applies the entailment rules until no new constraints are discovered. We have:

Proposition 4.1. *$\mathscr{C}$ and $\overline{\mathscr{C}}$ are equivalent.*

Verification (steps 2 and 3). In this phase the algorithm verifies that $\overline{\mathscr{C}}$ is satisfiable and fails if this is not the case. The key observation is that satisfiability of the type constraints does not depend upon *one particular* solution of the use constraints because the previous phase has computed all possible relations that must hold between use expressions. Therefore, we can independently verify the satisfiabilty of use and type constraints and fail if any of these check fails.

Recall that there is a finite number of use constraints, which are equations between use expressions made of a finite number of use variables ranging over a finite domain $\{0, 1, \omega\}$. Therefore, there exists a complete (albeit combinatorial) verification algorithm that determines whether or not the use constraints in $\overline{\mathscr{C}}$ are satisfiable. It is also possible to find an "optimal" algorithm that aims at maximizing the number of use variables that are assigned value 1 as opposed to ω. This algorithm can be improved in many ways, because the set of use constraints can be usually partitioned into many small sets of independent constraints, which can be efficiently solved in isolation. We do not discuss the issues related to solving use constraints any further.

If the use constraints in $\overline{\mathscr{C}}$ are satisfiable, then satisfiability of the type constraints is granted provided that there are no constraints relating types built with different constructors. For example, $\texttt{int} \leq_{\mathsf{c}} [\alpha]^{\kappa_1,\kappa_2}$ is clearly unsatisfiable. Because of [E-STRUCT]

and [E-TRANS], for any pair of types that must be related there is a constraint $t \sim_{\mathsf{c}} s$ in $\overline{\mathscr{C}}$. Therefore, if there is any such constraint where t and s are not type variables and are built using different topmost constructors, then $\overline{\mathscr{C}}$ is for sure unsatifiable and the algorithm fails.

Proposition 4.2. *If the algorithm **fail**s in this phase, then $\overline{\mathscr{C}}$ is not satisfiable.*

Synthesis (steps 4 and 5). The last phase computes a solution σ_t for the type constraints in $\sigma_u\overline{\mathscr{C}}$, which is the set of constraints $\overline{\mathscr{C}}$ where each use variable ρ has been replaced by $\sigma_u(\rho)$. To compute σ_t we need the functions below:

$$\mathsf{cls}_{\mathscr{R},\mathscr{C}}(T) \stackrel{\text{def}}{=} \{s \mid s\ \mathscr{R}_{\mathsf{c}}\ t \in \mathscr{C} \text{ for some } t \in T \text{ and } s \text{ is proper}\}$$

$$\mathsf{sup}_{\mathscr{C},\sigma}(T) \stackrel{\text{def}}{=} \begin{cases} [\mathsf{sup}_{\mathscr{C},\sigma}(\{t_i\}_{i\in I})]^{\bigvee_{i\in I}\sigma\kappa_i,\bigvee_{i\in I}\sigma\kappa_i'} & \text{if } \mathsf{cls}_{\leq,\mathscr{C}}(T) = \{[t_i]^{\kappa_i,\kappa_i'}\}_{i\in I} \neq \emptyset \\ \mathsf{sup}_{\mathscr{C},\sigma}(\{t_i\}_{i\in I}) \odot \mathsf{sup}_{\mathscr{C},\sigma}(\{s_i\}_{i\in I}) & \text{if } \mathsf{cls}_{\leq,\mathscr{C}}(T) = \{t_i \odot s_i\}_{i\in I} \neq \emptyset \\ \mathsf{zero}_{\mathscr{C},\sigma}(T) & \text{otherwise} \end{cases}$$

$$\mathsf{zero}_{\mathscr{C},\sigma}(T) \stackrel{\text{def}}{=} \begin{cases} [\mathsf{sup}_{\mathscr{C},\sigma}(\{t_i\}_{i\in I})]^{0,0} & \text{if } \mathsf{cls}_{\sim,\mathscr{C}}(T) = \{[t_i]^{\kappa_i,\kappa_i'}\}_{i\in I} \neq \emptyset \\ \mathsf{zero}_{\mathscr{C},\sigma}(\{t_i\}_{i\in I}) \odot \mathsf{zero}_{\mathscr{C},\sigma}(\{s_i\}_{i\in I}) & \text{if } \mathsf{cls}_{\sim,\mathscr{C}}(T) = \{t_i \odot s_i\}_{i\in I} \neq \emptyset \\ \mathtt{int} & \text{otherwise} \end{cases}$$

The set $\mathsf{cls}_{\mathscr{R},\mathscr{C}}(T)$ is made of the proper type expressions s such that $s\ \mathscr{R}_{\mathsf{c}}\ t \in \mathscr{C}$ for some $t \in T$. Note that not all $\mathscr{R}$'s are symmetric and that s is the type expression on the left hand side of $\mathscr{R}_{\mathsf{c}}$. So, $\mathsf{cls}_{\leq,\mathscr{C}}(T)$ is the set of type expressions that are lower bounds of some $t \in T$, while $\mathsf{cls}_{\sim,\mathscr{C}}(T)$ is the set of type expressions that are structurally compatible with some $t \in T$. Note also that $\mathsf{cls}_{\leq,\mathscr{C}}(T) \subseteq \mathsf{cls}_{\sim,\mathscr{C}}(T)$ because $\leq \subseteq \sim$. The algorithm (Table 4) resolves each variable α to $\mathsf{sup}_{\overline{\mathscr{C}},\sigma_u}(\{\alpha\})$ where, $\mathsf{sup}_{\mathscr{C},\sigma}(T)$ is, roughly speaking, the least upper bound of the types in T (even though the algorithm always invokes $\mathsf{sup}_{\mathscr{C},\sigma}$ with a singleton, in general we need to define $\mathsf{sup}_{\mathscr{C},\sigma}$ over a set of type expressions that are known to be equivalent). There are three cases that determine $\mathsf{sup}_{\mathscr{C},\sigma}(T)$: if there exists any lower bound for some of the types in T and these lower bounds are either channel or composite types, then $\mathsf{sup}_{\mathscr{C},\sigma}(T)$ is defined as the least upper bound of such lower bounds (first two cases in the definition of $\mathsf{sup}_{\mathscr{C},\sigma}$); if there is no lower bound but there exists at least one structural constraint involving any of the types in T, then $\mathsf{sup}_{\mathscr{C},\sigma}(T)$ is defined as a type that is structurally coherent with such constraints but has use 0 for all of its topmost channel types (third case in the definition of $\mathsf{sup}_{\mathscr{C},\sigma}$ and first two cases in the definition of $\mathsf{zero}_{\mathscr{C},\sigma}$); if there are no structural constraints involving any of the types in T, or if some of the types in T have been determined to be structurally coherent with `int`, then $\mathsf{zero}_{\mathscr{C},\sigma}(T)$ is defined to be `int` (third case in the definition of $\mathsf{zero}_{\mathscr{C},\sigma}$). An extension of our type system with polymorphism could further refine this case and leave type variables uninstantiated.

Interpreting $\mathsf{sup}_{\mathscr{C},\sigma}$ and $\mathsf{zero}_{\mathscr{C},\sigma}$ as functions is appropriate for presentation (and implementation) purposes, but formally imprecise for two reasons: **(1)** the equations given above are mutually dependent and **(2)** they are undefined for some particular T's (for instance, for $T = \{\mathtt{int}, [\mathtt{int}]^{0,0}\}$ which contains two types with incompatible structures

or for $T = \{[\mathtt{int}]^{\omega,0}, [\mathtt{int}]^{1,0}\}$ which contains two types with incompatible uses). Concerning **(1)**, the proper interpretation of the equations above is as a set $\{\alpha_i = t_i\}$ where each α_i has the form $\mathsf{sup}_{\mathscr{C},\sigma}(T)$ or $\mathsf{zero}_{\mathscr{C},\sigma}(T)$, $T \subseteq \mathsf{dom}(\mathscr{C})$, and t_i is determined by the right hand side of the equation. We know that this set is always finite because $\mathsf{dom}(\mathscr{C})$ is finite and so is its powerset. Furthermore, $\mathsf{zero}_{\mathscr{C},\sigma}$ always yields a proper type when it is defined and so does $\mathsf{sup}_{\mathscr{C},\sigma}$ when it is not defined in terms of $\mathsf{zero}_{\mathscr{C},\sigma}$. Therefore, the equations in $\{\alpha_i = t_i\}$ are contractive in the sense that there is no infinite chain of equations involving type variables only. In this case, it is known [1] that these equations can be folded into a possibly recursive contractive term using μ's. Concerning **(2)**, it turns out that when σ is a solution of the use constraints in $\mathscr{C}$ $\mathsf{sup}_{\mathscr{C},\sigma}(T)$ is defined if T is $\mathscr{C}$-composable and that $\mathsf{zero}_{\mathscr{C},\sigma}(T)$ is defined if T is $\mathscr{C}$-compatible, where we say that:

- T is *$\mathscr{C}$-composable* if $t \leq_{\mathsf{c}} s \in \mathscr{C}$ or $s \leq_{\mathsf{c}} t \in \mathscr{C}$ or $t \asymp_{\mathsf{c}} s \in \mathscr{C}$ for every $t, s \in T$;
- T is *$\mathscr{C}$-compatible* if $t \sim_{\mathsf{c}} s \in \mathscr{C}$ for every $t, s \in T$.

Indeed observe that: $\mathsf{cls}_{\leq,\mathscr{C}}(T)$ is $\mathscr{C}$-composable if so is T by [E-COMP 2]; $\mathsf{cls}_{\sim,\mathscr{C}}(T)$ is $\mathscr{C}$-compatible if T is $\mathscr{C}$-composable by [E-STRUCT]; if $\{[t_i]^{\kappa_i,\kappa_i'}\}_{i\in I}$ is $\mathscr{C}$-composable then the least upper bounds $\bigvee_{i\in I}\sigma\kappa_i$ and $\bigvee_{i\in I}\sigma\kappa_i'$ are defined (consequence of the use constraints generated by [E-CHANNEL] and the hypothesis that σ is a solution of them); if $\{[t_i]^{\kappa_i,\kappa_i'}\}_{i\in I}$ is $\mathscr{C}$-compatible, then $\{t_i\}_{i\in I}$ is $\mathscr{C}$-composable (consequence of the $=_{\mathsf{c}}$ constraints generated by [E-CHANNEL]); if $\{t_i \odot s_i\}_{i\in I}$ is $\mathscr{C}$-composable/compatible, then so are the sets $\{t_i\}_{i\in I}$ and $\{s_i\}_{i\in I}$ by [E-OPER]; the type expressions in $\mathscr{C}$-composable/compatible sets are built using the same topmost constructor (check in step 3 of the algorithm). Finally, observe that a singleton $\{\alpha\}$ is always $\mathscr{C}$-composable, so the invocations of $\mathsf{sup}_{\overline{\mathscr{C}},\sigma_u}$ in Table 4 regard well-defined equations. We conclude:

Theorem 4.1 (correctness). *If the algorithm returns σ, then σ is a solution for $\mathscr{C}$.*

Each step of the algorithm terminates and if the algorithm fails it is because $\mathscr{C}$ has no solution (Proposition 4.2). Therefore:

Corollary 4.1 (completeness). *If $\mathscr{C}$ is satisfiable, the algorithm returns a solution.*

Example 4.1. The saturation of the constraint set $\mathscr{C}$ computed in Example 3.1 contains, among others, the constraints $[\mathtt{int}]^{0,\rho_1} \times \gamma \asymp_{\mathsf{c}} \beta \times [\mathtt{int}]^{0,\rho_2}$ and consequently $[\mathtt{int}]^{0,\rho_1} \asymp_{\mathsf{c}} \beta$ and $\gamma \asymp_{\mathsf{c}} [\mathtt{int}]^{0,\rho_2}$ by [E-COMP 1] and [E-COMP 2]. An optimal solution of the use constraints in $\overline{\mathscr{C}}$ is $\sigma_u \stackrel{\text{def}}{=} \{\rho_1 \mapsto 1, \rho_2 \mapsto 1, \rho_3 \mapsto 1\}$. From this we obtain

$$\mathsf{sup}_{\sigma_u\overline{\mathscr{C}}}(\{\alpha\}) = [\mathtt{int}]^{0,1} \times [\mathtt{int}]^{0,1}$$

indicating that the pair of channels received from a is shared by the two `let` processes in such a way that each of the two channel contained therein is used exactly once. ■

5 Concluding Remarks

Previous works on the linear π-calculus either ignore composite types [9,7] or are based on an interpretation of linearity that limits data sharing and parallelism [5,6]. Recursive

types have also been neglected, despite their prominent role for describing complex interactions occurring on linear channels [2]. In this work we extend the linear π-calculus with both composite and recursive types and we adopt a more relaxed attitude towards linearity that fosters data sharing and parallelism while preserving full type reconstruction. 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 more general type composition operator. This small change has nonetheless non-trivial consequences on the reconstruction algorithm, which must reconcile the propagation of constraints across composite types with the impossibility to rely on plain type unification due to the fact that different occurrences of the same identifier may be assigned different types and because of recursive types. Technically, we tackle this problem by expressing type composition (which is a ternary relation $t_1 + t_2 = t_3$) in terms of two simpler binary relations, namely compatibility $t_1 \asymp t_2$ and order $t_i \leq t_3$. 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 can only be passed around in messages without actually ever being used. That is, they can be erased without affecting processes. In our type system, it is the existence of these types that enables the sharing of composite values (see the decomposition of t_{list} into t_{even} and t_{odd}).

We have implemented a prototype based on the naïve constraint saturation and combinatorial verification of use expressions described in Section 4. Support for polymorhism and more clever implementations of the type reconstruction algorithm are obvious yet important subjects for future work.

References

1. B. Courcelle. Fundamental properties of infinite trees. *Theor. Comp. Sci.*, 25:95–169, 1983.
2. O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In *PPDP'12*, pages 139–150. ACM, 2012.
3. K. Honda. Types for dyadic interaction. In *CONCUR'93*, LNCS 715, pages 509–523. Springer, 1993.
4. 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.
5. 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/`.
6. A. Igarashi and N. Kobayashi. Type-based analysis of communication for concurrent programming languages. In *SAS'97*, LNCS 1302, pages 187–201. Springer, 1997.
7. A. Igarashi and N. Kobayashi. Type Reconstruction for Linear π-Calculus with I/O Subtyping. *Inf. and Comp.*, 161(1):1–44, 2000.
8. 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`.
9. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999.
10. 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.
11. D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *FPCA'95*, pages 1–11, 1995.

A Supplement to Section 2

Lemma A.1. *If $\Gamma \vdash P$ and $\Gamma' \asymp \Gamma'$ and $\Gamma + \Gamma'$ is defined, then $\Gamma + \Gamma' \vdash P$.*

Lemma A.2. *If $\Gamma \vdash P$ and $P \equiv Q$, then $\Gamma \vdash Q$.*

Proof. The rule relies on the associativity of $+$. We show two interesting cases only.

[S-PAR 1] Then $P = \mathbf{0} \mid Q$. From [T-PAR] we deduce $\Gamma = \Gamma_1 + \Gamma_2$ and $\Gamma_1 \vdash \mathbf{0}$ and $\Gamma_2 \vdash Q$. From [T-IDLE] we deduce $\Gamma_1 \asymp \Gamma_1$. We conclude by Lemma A.1.

[S-REP] Then $P = *P' \equiv *P' \mid P' = Q$. From the hypothesis $\Gamma \vdash P$ we deduce $\Gamma = \Gamma' + \Gamma'$ for some Γ' such that $\Gamma' \vdash P'$. We conclude with an application of [T-PAR] and observing that $\Gamma = \Gamma' + \Gamma' = \Gamma' + \Gamma' + \Gamma' = \Gamma + \Gamma'$. □

Lemma A.3 (substitution). *If $\Gamma, x : t \vdash P$ and $\Gamma' \vdash \mathsf{v} : t$ and $\Gamma + \Gamma'$ is defined, then $\Gamma + \Gamma' \vdash P\{\mathsf{v}/x\}$.*

Let $\xrightarrow{\ell}$ be the least relation between type environments such that

$$\Gamma, a : [t]^{1,1} \xrightarrow{a} \Gamma, a : [t]^{0,0} \quad \text{and} \quad \Gamma, a : [t]^{\omega,\omega} \xrightarrow{a} \Gamma, a : [t]^{\omega,\omega} \quad \text{and} \quad \Gamma \xrightarrow{\tau} \Gamma$$

We say that Γ is *balanced* if every type $[t]^{\kappa_1,\kappa_2}$ in the range of Γ where $0 < \kappa_1$ and $0 < \kappa_2$ is such that $\kappa_1 = \kappa_2$.

Proposition A.1. *The following properties hold:*

1. *If Γ is balanced and $\Gamma \xrightarrow{\ell} \Gamma'$, then Γ' is balanced.*
2. *If $\Gamma \xrightarrow{\ell} \Gamma'$ and $\Gamma + \Gamma''$ is defined, then $\Gamma + \Gamma'' \xrightarrow{\ell} \Gamma' + \Gamma''$.*

Theorem A.1 (subject reduction). *Let $\Gamma \vdash P$ and Γ balanced 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 $P = a!\langle \mathsf{v} \rangle \mid a?(x).R$ and $\ell = a$. From [T-PAR] we deduce $\Gamma = \Gamma_1 + \Gamma_2$ where $\Gamma_1 \vdash a!\langle \mathsf{v} \rangle$ and $\Gamma_2 \vdash a?(x).R$. From [T-OUT] we deduce $\Gamma_1 = \Gamma_1' + a : [t]^{0,\kappa}$ and $0 < \kappa$ and $\Gamma_1' \vdash \mathsf{v} : t$. From [T-IN] we deduce $\Gamma_2 = \Gamma_2' + a : [t]^{\kappa,0}$ and $\Gamma_2', x : t \vdash R$. The reason why we know that we have the same t and κ in Γ_1 and Γ_2 comes from the definition of $+$ and the fact that these two environments were combined together in Γ which is balanced. Note that $\Gamma_1' + \Gamma_2'$ is defined and $\Gamma \xrightarrow{a} \Gamma_1' + \Gamma_2'$. From Lemma A.3 we conclude $\Gamma_1' + \Gamma_2' \vdash R\{\mathsf{v}/x\}$.

[R-LET] Then $P = \mathtt{let}\ x, y = \mathsf{v}, \mathsf{w}\ \mathtt{in}\ R \xrightarrow{\tau} R\{\mathsf{v}, \mathsf{w}/x, y\} = Q$. From [T-LET] we deduce $\Gamma = \Gamma_1 + \Gamma_2$ and $\Gamma_1 \vdash \mathsf{v}, \mathsf{w} : t \times s$ and $\Gamma_2, x : t, y : s \vdash R$. From [T-PAIR] we deduce $\Gamma_1 = \Gamma_{11} + \Gamma_{12}$ and $\Gamma_{11} \vdash \mathsf{v} : t$ and $\Gamma_{12} \vdash \mathsf{w} : s$. We conclude $\Gamma \vdash Q$ by Lemma A.3.

[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 $\Gamma_i \vdash P_i$ where Γ_i is balanced for $i \in \{1, 2\}$. 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.1 we deduce that $\Gamma \xrightarrow{\ell} \Gamma_1' + \Gamma_2$. We conclude $\Gamma' \vdash Q$ by taking $\Gamma' = \Gamma_1' + \Gamma_2$. □

Note that Theorem A.1 establishes not only a subject reduction result, but also a soundness result because it says that a channel is used no more than what is allowed by its type. It is also possible to establish some basic safety properties, in particular that well-typed `let`'s and `case`'s always reduce. The proof is easy, we omit the details.

B Supplement to Section 3

We say that a solution σ for $\mathscr{C}$ is *minimal* if $t\sigma = \sum_{s \leq_c t \in \mathscr{C}} s\sigma$.

Lemma B.1. *If $\Gamma_1 \sqcup \Gamma_2 \rightsquigarrow \Gamma; \mathscr{C}$ and σ is a solution for $\mathscr{C}_0 \supseteq \mathscr{C}$, then $\Gamma_1\sigma + \Gamma_2\sigma = \Gamma\sigma$.*

Proof. By induction on the derivation of $\Gamma_1 \sqcup \Gamma_2 \rightsquigarrow \Gamma; \mathscr{C}$.

$\boxed{\Gamma_i = \emptyset \text{ for } i \in \{1,2\}}$ Then $\Gamma = \Gamma_{3-i}$ and we conclude $\Gamma\sigma = \Gamma_{3-i}\sigma = \Gamma_1\sigma + \Gamma_2\sigma$.

$\boxed{\Gamma_1 = \Gamma_1', u : t \text{ and } \Gamma_2 = \Gamma_2', u : s}$ Then $\Gamma_1' \sqcup \Gamma_2' \rightsquigarrow \Gamma'; \mathscr{C}'$ and $\Gamma = \Gamma', u : \alpha$ and $\mathscr{C} = \mathscr{C}' \cup \{t \asymp_c s, t \leq_c \alpha, s \leq_c \alpha\}$. If σ is a solution for $\mathscr{C}_0$, we deduce that $t\sigma + s\sigma = \sigma(\alpha)$. By induction hypothesis we deduce $\Gamma_1'\sigma + \Gamma_2'\sigma = \Gamma'\sigma$. We conclude $\Gamma_1\sigma + \Gamma_2\sigma = \Gamma\sigma$. □

Lemma B.2. *If $\Gamma_1 \sqcap \Gamma_2 \rightsquigarrow \Gamma; \mathscr{C}$ and σ is a solution for $\mathscr{C}_0 \supseteq \mathscr{C}$, then $\Gamma_1\sigma = \Gamma_2\sigma = \Gamma\sigma$.*

Proof. Immediate consequence of the definition of $\Gamma_1 \sqcap \Gamma_2 \rightsquigarrow \Gamma; \mathscr{C}$. □

Lemma B.3. *If $e : t \triangleright \Gamma; \mathscr{C}$ and σ is a solution of $\mathscr{C}_0 \supseteq \mathscr{C}$, then $\Gamma\sigma \vdash e : t\sigma$.*

Proof. By induction on the derivation of $e : t \triangleright \Gamma; \mathscr{C}$ and by cases on the last rule applied. We only show the case of [I-PAIR], the other cases being either trivial or similar. We have $e = e_1, e_2$ and $t = t_1 \times t_2$ and $\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3$ where $\Gamma_1 \sqcup \Gamma_2 \rightsquigarrow \Gamma; \mathscr{C}_3$ and $e_i : t_i \triangleright \Gamma_i; \mathscr{C}_i$ for $i = 1,2$. By induction hypothesis we deduce $\Gamma_i\sigma \vdash e : t_i\sigma$ for $i = 1,2$. From Lemma B.1 we obtain $\Gamma_1\sigma + \Gamma_2\sigma = \Gamma\sigma$. We conclude with an application of [T-PAIR]. □

Theorem B.1. *If $P \triangleright \Gamma; \mathscr{C}$ and σ is a solution for $\mathscr{C}_0 \supseteq \mathscr{C}$, then $\Gamma\sigma \vdash P$.*

Proof. By induction on the derivation of $P \triangleright \Gamma; \mathscr{C}$ and by cases on the last rule applied.

$\boxed{\text{[I-IDLE]}}$ Trivial.

$\boxed{\text{[I-IN]}}$ Then $P = u?(x).Q$ and $Q \triangleright \Gamma', x : t; \mathscr{C}_1$ and $\Gamma' \sqcup u : [t]^{\rho,0} \rightsquigarrow \Gamma; \mathscr{C}_2$ and $\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \{0 <_c \rho\}$. By induction hypothesis we deduce $\Gamma'\sigma, x : t\sigma \vdash Q$. By Lemma B.1 we deduce $\Gamma'\sigma + u : [t\sigma]^{\sigma(\rho),0} = \Gamma\sigma$. From the hypothesis that σ is a solution for $\mathscr{C}_0$ we know $0 < \sigma(\rho)$. We conclude with an application of [T-IN].

$\boxed{\text{[I-OUT]}}$ Then $P = u!\langle e \rangle$ and $e : t \triangleright \Gamma'; \mathscr{C}_1$ and $\Gamma' \sqcup u : [t]^{0,\rho} \rightsquigarrow \Gamma; \mathscr{C}_2$ and $\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \{0 <_c \rho\}$. From Lemma B.3 we deduce $\Gamma'\sigma \vdash e : t\sigma$. From Lemma B.1 we deduce $\Gamma'\sigma + u : [t\sigma]^{0,\sigma(\rho)} = \Gamma\sigma$. From the hypothesis that σ is a solution of $\mathscr{C}_0$ we know $0 < \sigma(\rho)$. We conclude with an application of [T-OUT].

$\boxed{\text{[I-PAR]}}$ Then $P = P_1 \mid P_2$ and $P_i \triangleright \Gamma_i; \mathscr{C}_i$ for $i = 1,2$ and $\Gamma_1 \sqcup \Gamma_2 \rightsquigarrow \Gamma; \mathscr{C}_3$ and $\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3$. By induction hypothesis we deduce $\Gamma_i\sigma \vdash P_i$ for $i = 1,2$. By Lemma B.1 we deduce $\Gamma_1\sigma + \Gamma_2\sigma = \Gamma\sigma$. We conclude with an application of [T-PAR].

[I-REP] Then $P = *Q$ and $Q \triangleright \Gamma'; \mathscr{C}_1$ and $\Gamma' \sqcup \Gamma' \rightsquigarrow \Gamma; \mathscr{C}_2$ and $\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2$. By induction hypothesis we deduce $\Gamma'\sigma \vdash Q$. By Lemma B.1 we deduce $\Gamma'\sigma + \Gamma'\sigma = \Gamma\sigma$. We conclude with an application of [T-REP].

[I-NEW] Then $P = (\nu a)Q$ and $Q \triangleright \Gamma, a : [t]^{\kappa_1,\kappa_2}; \mathscr{C}'$ and $\mathscr{C} = \mathscr{C}' \cup \{\kappa_1 =_{\mathsf{c}} \kappa_2\}$. By induction hypothesis we deduce $\Gamma\sigma, a : [t\sigma]^{\kappa_1\sigma,\kappa_2\sigma} \vdash Q$. Since σ is a solution of $\mathscr{C}_0$ we know that $\kappa_1\sigma = \kappa_2\sigma$. We conclude with an application of [T-NEW].

[I-WEAK] Then $\Gamma = \Gamma', u : \alpha$ and $\mathscr{C} = \mathscr{C}' \cup \{\alpha \asymp_{\mathsf{c}} \alpha\}$ and $P \triangleright \Gamma'; \mathscr{C}'$. By induction hypothesis we deduce $\Gamma'\sigma \vdash P$. Since σ is a solution of $\mathscr{C}_0$ we know that $\sigma(\alpha) \asymp \sigma(\alpha)$. Since $u \notin \mathsf{dom}(\Gamma')$ we know that $\Gamma'\sigma + u : \sigma(\alpha)$ is defined. By Lemma A.1 we conclude $\Gamma'\sigma, u : \sigma(\alpha) \vdash P$.

[I-LET] Then $P = \mathtt{let}\ x,y = e\ \mathtt{in}\ Q$ and $e : t \triangleright \Gamma_1; \mathscr{C}_1$ and $Q \triangleright \Gamma_2, x : t_1, y : t_2; \mathscr{C}_2$ and $\Gamma_1 \sqcup \Gamma_2 \rightsquigarrow \Gamma; \mathscr{C}_3$ and $\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \{t =_{\mathsf{c}} t_1 \times t_2\}$. By Lemma B.3 we deduce $\Gamma_1\sigma \vdash e : t\sigma$. By induction hypothesis we deduce $\Gamma_2\sigma, x : t_1\sigma, y : t_2\sigma \vdash Q$. Since σ is a solution of $\mathscr{C}_0$ we know that $t\sigma = t_1\sigma \times t_2\sigma$. We conclude with an application of [T-LET].

[I-CASE] Then $P = \mathtt{case}\ e\ \mathtt{of}\{i\ x_i \Rightarrow P_i\}_{i=\mathtt{inl},\mathtt{inr}}$ and $e : t \triangleright \Gamma_1; \mathscr{C}_1$ and $P_i \triangleright \Gamma_i, x_i : t_i; \mathscr{C}_i$ for $i = \mathtt{inl}, \mathtt{inr}$ and $\Gamma_{\mathtt{inl}} \sqcap \Gamma_{\mathtt{inr}} \rightsquigarrow \Gamma_2; \mathscr{C}_2$ and $\Gamma_1 \sqcup \Gamma_2 \rightsquigarrow \Gamma; \mathscr{C}_3$ and $\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \{t =_{\mathsf{c}} t_{\mathtt{inl}} \oplus t_{\mathtt{inr}}\}$. By Lemma B.3 we deduce $\Gamma_1\sigma \vdash e : t\sigma$. By induction hypothesis we deduce $\Gamma_i\sigma \vdash P_i$ for $i = \mathtt{inl}, \mathtt{inr}$. By Lemma B.2 we deduce $\Gamma_{\mathtt{inl}}\sigma = \Gamma_{\mathtt{inr}}\sigma = \Gamma_2\sigma$. By Lemma B.1 we deduce $\Gamma_1\sigma + \Gamma_2\sigma = \Gamma\sigma$. We conclude with an application of [T-CASE]. □

A judgment Γ is well formed if every channel in $\mathsf{dom}(\Gamma)$ has a channel type. We make the implicit assumption that all type environments are well formed.

Lemma B.4. *If $\Gamma_1\sigma+\Gamma_2\sigma$ is defined, then there exist Γ, $\mathscr{C}$, and $\sigma'\supseteq\sigma$ such that $\Gamma_1\sqcup\Gamma_2\rightsquigarrow\Gamma;\mathscr{C}$ and $\Gamma_1\sigma'+\Gamma_2\sigma'=\Gamma\sigma'$ and σ' is solution of $\mathscr{C}$.*

Proof. By induction on Γ_1 and Γ_2.

$\boxed{\Gamma_i=\emptyset \text{ for } i\in\{1,2\}}$ Take $\Gamma=\Gamma_{3-i}$, $\mathscr{C}=\emptyset$, and $\sigma'=\sigma$. We conclude by observing that $\Gamma_1\sqcup\Gamma_2\rightsquigarrow\Gamma_{3-i};\emptyset$.

$\boxed{\Gamma_1=\Gamma_1',u:t \text{ and } \Gamma_2=\Gamma_2',u:s}$ Since $\Gamma_1\sigma+\Gamma_2\sigma$ is defined, we know that $\Gamma_1'\sigma+\Gamma_2'\sigma$ and $t\sigma+s\sigma$ are also defined. By induction hypothesis we deduce that there exist Γ', $\mathscr{C}'$, and $\sigma''\supseteq\sigma$ such that $\Gamma_1'\sqcup\Gamma_2'\rightsquigarrow\Gamma';\mathscr{C}'$ and $\Gamma_1'\sigma''+\Gamma_2'\sigma''=\Gamma'\sigma''$ and σ'' is a solution of $\mathscr{C}'$. Take $\Gamma=\Gamma',u:\alpha$ where α is fresh, $\mathscr{C}=\mathscr{C}'\cup\{t\asymp_{\mathsf{c}}s,t\leq_{\mathsf{c}}\alpha,s\leq_{\mathsf{c}}\alpha\}$ and $\sigma'=\sigma''\cup\{\alpha\mapsto t\sigma+s\sigma\}$. We conclude by observing that σ' is a solution of $\mathscr{C}$. □

Theorem B.2 (principal typing). *If $\Gamma\vdash P$, then there exist Γ_0, $\mathscr{C}_0$ and σ that is a solution of $\mathscr{C}_0$ such that $P\triangleright\Gamma_0;\mathscr{C}_0$ and $\Gamma=\Gamma_0\sigma$.*

Proof. By induction on the derivation of $\Gamma\vdash P$ and by cases on the last rule applied.

$\boxed{\text{[T-IDLE]}}$ Then $P=\mathbf{0}$ and $\Gamma\asymp\Gamma$. Let $\Gamma=\{u_i:t_i\}_{i\in I}$ where $I=\{1,\ldots,n\}$. We take $\Gamma_0=\{u_i:\alpha_i\}_{i\in I}$ and $\mathscr{C}_0=\{\alpha_i\asymp_{\mathsf{c}}\alpha_i\}$ and $\sigma=\{\alpha_i=t_i\}_{i\in I}$ where the α_i's are all fresh variables. By repeated applications of [I-WEAK] and one application of [I-IDLE] we derive $\mathbf{0}\triangleright\Gamma_0;\mathscr{C}_0$. We conclude by observing that σ is a solution of $\mathscr{C}_0$ and $\Gamma=\Gamma_0\sigma$.

$\boxed{\text{[T-IN]}}$ Then $P=u?(x).Q$ and $\Gamma',x:t\vdash Q$ and $\Gamma=\Gamma'+u:[t]^{\kappa,0}$ and $0<\kappa$. By induction hypothesis we deduce that there exist Γ_0', t_0', $\mathscr{C}_0'$, and σ' that is a solution of $\mathscr{C}_0'$ and $Q\triangleright\Gamma_0',x:t_0';\mathscr{C}_0'$ and $\Gamma'=\Gamma_0'\sigma'$ and $t=t_0'\sigma'$. Let $\sigma''=\sigma'\cup\{\rho\mapsto\kappa\}$ where ρ is fresh. By Lemma B.4 we deduce that there exist Γ_0, $\mathscr{C}$, and $\sigma\supseteq\sigma''$ such that $\Gamma_0'\sqcup u:[t_0']^{\kappa,0}\rightsquigarrow\Gamma_0;\mathscr{C}$ and $\Gamma_0'\sigma+u:[t_0'\sigma]^{\kappa,0}=\Gamma_0\sigma=\Gamma$ and σ is a solution of $\mathscr{C}$. We conclude $P\triangleright\Gamma_0;\mathscr{C}_0$ with an application of [I-IN] by taking $\mathscr{C}_0=\mathscr{C}_0'\cup\mathscr{C}\cup\{0<_{\mathsf{c}}\rho\}$ and by observing that σ is a solution of $\mathscr{C}_0$.

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

$\boxed{\text{[T-REP]}}$ Then $P=*Q$ and $\Gamma'\vdash Q$ and $\Gamma=\Gamma'+\Gamma'$. By induction hypothesis we deduce that there exist Γ_0', $\mathscr{C}_0'$, and σ' that is a minimal solution of $\mathscr{C}_0'$ such that $Q\triangleright\Gamma_0';\mathscr{C}_0'$ and $\Gamma'=\Gamma_0'\sigma'$. By Lemma B.4 we deduce that there exist Γ_0, $\mathscr{C}'$, $\sigma\supseteq\sigma'$ such that $\Gamma_0'\sqcup\Gamma_0'\rightsquigarrow\Gamma_0;\mathscr{C}'$ and $\Gamma_0'\sigma+\Gamma_0'\sigma=\Gamma_0\sigma=\Gamma$ and σ is a solution of $\mathscr{C}'$. We conclude $P\triangleright\Gamma_0;\mathscr{C}$ with an application of [I-REP] by taking $\mathscr{C}=\mathscr{C}_0'\cup\mathscr{C}'$.

C Supplement to Section 4

Definition C.1. *We say that $\mathscr{C}$ is* closed for compatibility *if for every $t, s_1, s_2 \in \mathsf{dom}(\overline{\mathscr{C}})$ and $s_i \leq_{\mathsf{c}} t \in \overline{\mathscr{C}}$ for $i = 1,2$ we have either $s_1 \leq_{\mathsf{c}} s_2$ or $s_2 \leq_{\mathsf{c}} s_1$ or $s_1 \asymp_{\mathsf{c}} s_2$.*

Proposition C.1. *Let $\mathscr{C}$ be a set of constraints generated by the reconstruction system in Table 3. Then $\mathscr{C}$ is closed for compatibility.*

Proof. Follows by a simple analysis of the sets of constraints generated by reconstruction system. □

Proposition C.2. *If $\mathscr{C}$ is closed for compatibility, then so is $\overline{\mathscr{C}}$.*

Proof. Consequence of the hypothesis that $\mathscr{C}$ is closed for compatibility and the fact that $\overline{\mathscr{C}}$ is closed under [E-COMP 2]. □

Proposition C.3 (Proposition 4.1). *$\mathscr{C}$ and $\overline{\mathscr{C}}$ are equivalent.*

Proof. Follows immediately from the saturation laws [E-*]. □

Lemma C.1. *Let $\emptyset \neq T \subseteq S$ where S is composable. The following properties hold:*

1. $\mathsf{cls}_{\sim,\mathscr{C}}(T,\mathscr{C}) = \mathsf{cls}_{\sim,\mathscr{C}}(S,\mathscr{C})$;
2. *If $t, s \in S$ implies $t =_{\mathsf{c}} s \in \mathscr{C}$, then $\mathsf{cls}_{\mathscr{R},\mathscr{C}}(T,\mathscr{C}) = \mathsf{cls}_{\mathscr{R},\mathscr{C}}(S,\mathscr{C})$ for every $\mathscr{R} \in \{\leq, \sim\}$.*

Proof. Easy consequences of the definition of $\mathsf{cls}_{\mathscr{R},\mathscr{C}}$. □

Lemma C.2. *Let $\mathscr{C}$ be a saturated set of constraints that has passed the checks at steps 2 and 3 of the algorithm in Table 4 and whose use constraints are solved. Then $\mathsf{def}(\{t\},\mathscr{C})\ \mathscr{R}\ \mathsf{def}(\{s\},\mathscr{C})$ for every $t\ \mathscr{R}_{\mathsf{c}}\ s \in \mathscr{C}$.*

Proof. In this proof we write $T\ \mathscr{R}_{\mathsf{c}}\ S \in \mathscr{C}$ if $t\ \mathscr{R}_{\mathsf{c}}\ s \in \mathscr{C}$ for every $t \in T$ and $s \in S$. Let

$$\begin{aligned}\mathscr{C}(\leq) \stackrel{\text{def}}{=}\ & \{(\mathsf{def}(T,\mathscr{C}), \mathsf{def}(S,\mathscr{C})) \mid \emptyset \neq T \subseteq S \subseteq \mathsf{dom}(\mathscr{C}) \text{ and } S \text{ is composable}\} \\ & \cup \{(\mathsf{zero}_{\mathscr{C},\sigma}(T,\mathscr{C}), \mathsf{def}(S,\mathscr{C})) \mid \emptyset \neq T, S \subseteq \mathsf{dom}(\mathscr{C}) \text{ and } T \sim_{\mathsf{c}} S \text{ and } S \text{ is composable}\} \\ & \cup \{(\mathsf{zero}_{\mathscr{C},\sigma}(T,\mathscr{C}), \mathsf{zero}_{\mathscr{C},\sigma}(S,\mathscr{C})) \mid \emptyset \neq T, S \subseteq \mathsf{dom}(\mathscr{C}) \text{ and } T \sim_{\mathsf{c}} S\}\end{aligned}$$

$$\begin{aligned}\mathscr{C}(\asymp) \stackrel{\text{def}}{=}\ & \{(\mathsf{def}(T,\mathscr{C}), \mathsf{def}(S,\mathscr{C})) \mid \emptyset \neq T, S \subseteq \mathsf{dom}(\mathscr{C}) \text{ and } T \asymp_{\mathsf{c}} S \in \mathscr{C}\} \\ & \cup \{(\mathsf{zero}_{\mathscr{C},\sigma}(T,\mathscr{C}), \mathsf{def}(S,\mathscr{C})) \mid \emptyset \neq T, S \subseteq \mathsf{dom}(\mathscr{C}) \text{ and } T \asymp_{\mathsf{c}} S \in \mathscr{C}\} \\ & \cup \{(\mathsf{def}(T,\mathscr{C}), \mathsf{zero}_{\mathscr{C},\sigma}(S,\mathscr{C})) \mid \emptyset \neq T, S \subseteq \mathsf{dom}(\mathscr{C}) \text{ and } T \asymp_{\mathsf{c}} S \in \mathscr{C}\} \\ & \cup \{(\mathsf{zero}_{\mathscr{C},\sigma}(T,\mathscr{C}), \mathsf{zero}_{\mathscr{C},\sigma}(S,\mathscr{C})) \mid \emptyset \neq T, S \subseteq \mathsf{dom}(\mathscr{C}) \text{ and } T \asymp_{\mathsf{c}} S \in \mathscr{C}\}\end{aligned}$$

and observe that $t\ \mathscr{R}_{\mathsf{c}}\ s \in \mathscr{C}$ implies $(\mathsf{def}(\{t\},\mathscr{C}), \mathsf{def}(\{s\},\mathscr{C})) \in \mathscr{C}(\mathscr{R})$ by definition of $\mathscr{C}(\mathscr{R})$. It is enough to prove that $\mathscr{C}(\mathscr{R}) \subseteq \mathscr{R}$.

Regarding the case $\mathscr{R} = \leq$, let $(t,s) \in \mathscr{C}(\leq)$. Then there exist T and S such that $\emptyset \neq T \subseteq S$ and S is composable and $t = \mathsf{def}(T,\mathscr{C})$ and $s = \mathsf{def}(S,\mathscr{C})$. We analyze the possible cases, taking into account Lemma C.1(1).

- $t = [t']^{\bigvee_{i\in I}\kappa_i,\bigvee_{i\in I}\kappa'_i}$ and $s = [s']^{\bigvee_{j\in J}\kappa''_i,\bigvee_{j\in J}\kappa'''_j}$ where $\mathsf{cls}_{\leq,\mathscr{C}}(T,\mathscr{C}) = \{[t_i]^{\kappa_i,\kappa'_i}\}_{i\in I}$ and $\mathsf{cls}_{\leq,\mathscr{C}}(S,\mathscr{C}) = \{[s_j]^{\kappa''_j,\kappa'''_j}\}_{j\in J}$ and $\emptyset \neq I \subseteq J$. We deduce $\bigvee_{i\in I}\kappa_i \leq \bigvee_{j\in J}\kappa''_j$ and $\bigvee_{i\in I}\kappa'_i \leq \bigvee_{j\in J}\kappa'''_j$. We have $t_i =_{\mathsf{c}} s_j \in \mathscr{C}$ for every $i \in I$ and $j \in J$, therefore $t' = s'$ by Lemma C.1(2). We conclude $t \leq s$.
- $t = [t']^{0,0}$ and $s = [s']^{\bigvee_{j\in J}\kappa''_j,\bigvee_{j\in J}\kappa'''_j}$ where $\mathsf{cls}_{\sim,\mathscr{C}}(T,\mathscr{C}) = \{[t_i]^{\kappa_i,\kappa'_i}\}_{i\in I}$ and $\mathsf{cls}_{\leq,\mathscr{C}}(S,\mathscr{C}) = \{[s_j]^{\kappa''_j,\kappa'''_j}\}_{j\in J}$ and $\emptyset \neq I,J$. We conclude $t \leq s$ using Lemma C.1(2) like in the previous case.
- $t = [t']^{0,0}$ and $s = [s']^{0,0}$ where $\mathsf{cls}_{\sim,\mathscr{C}}(T,\mathscr{C}) = \{[t_i]^{\kappa_i,\kappa'_i}\}_{i\in I}$ and $\mathsf{cls}_{\sim,\mathscr{C}}(S,\mathscr{C}) = \{[s_j]^{\kappa''_j,\kappa'''_j}\}_{j\in J}$ and $\emptyset \neq I,J$. We conclude $t = s$ using Lemma C.1(2) like in the previous cases.
- $t = t_1 \odot t_2$ and $s = s_1 \odot s_2$ where $t_1 = \mathsf{def}(\{t_i\}_{i\in I},\mathscr{C})$ and $t_2 = \mathsf{def}(\{t'_i\}_{i\in I},\mathscr{C})$ and $s_1 = \mathsf{def}(\{s_j\}_{j\in J},\mathscr{C})$ and $s_2 = \mathsf{def}(\{s'_j\}_{j\in J},\mathscr{C})$ and $\mathsf{cls}_{\leq,\mathscr{C}}(T,\mathscr{C}) = \{t_i \odot t'_i\}_{i\in I}$ and $\mathsf{cls}_{\leq,\mathscr{C}}(S,\mathscr{C}) = \{s_j \odot s'_j\}_{j\in J}$ and $\emptyset \neq I \subseteq J$. We conclude $(t_1,s_1) \in \mathscr{C}(\leq)$ and $(t_2,s_2) \in \mathscr{C}(\leq)$ by definition of $\mathscr{C}(\leq)$.
- $t = t_1 \odot t_2$ and $s = s_1 \odot s_2$ where $t_1 = \mathsf{zero}_{\mathscr{C},\sigma}(\{t_i\}_{i\in I},\mathscr{C})$ and $t_2 = \mathsf{zero}_{\mathscr{C},\sigma}(\{t'_i\}_{i\in I},\mathscr{C})$ and $s_1 = \mathsf{def}(\{s_j\}_{j\in J},\mathscr{C})$ and $s_2 = \mathsf{def}(\{s'_j\}_{j\in J},\mathscr{C})$ and $\mathsf{cls}_{\sim,\mathscr{C}}(T,\mathscr{C}) = \{t_i \odot t'_i\}_{i\in I}$ and $\mathsf{cls}_{\leq,\mathscr{C}}(S,\mathscr{C}) = \{s_j \odot s'_j\}_{j\in J}$ and $\emptyset \neq I,J$. We conclude $(t_1,s_1) \in \mathscr{C}(\leq)$ and $(t_2,s_2) \in \mathscr{C}(\leq)$ by definition of $\mathscr{C}(\leq)$.
- $t = t_1 \odot t_2$ and $s = s_1 \odot s_2$ where $t_1 = \mathsf{zero}_{\mathscr{C},\sigma}(\{t_i\}_{i\in I},\mathscr{C})$ and $t_2 = \mathsf{zero}_{\mathscr{C},\sigma}(\{t'_i\}_{i\in I},\mathscr{C})$ and $s_1 = \mathsf{zero}_{\mathscr{C},\sigma}(\{s_j\}_{j\in J},\mathscr{C})$ and $s_2 = \mathsf{zero}_{\mathscr{C},\sigma}(\{s'_j\}_{j\in J},\mathscr{C})$ and $\mathsf{cls}_{\sim,\mathscr{C}}(T,\mathscr{C}) = \{t_i \odot t'_i\}_{i\in I}$ and $\mathsf{cls}_{\sim,\mathscr{C}}(S,\mathscr{C}) = \{s_j \odot s'_j\}_{j\in J}$ and $\emptyset \neq I,J$. We conclude $(t_1,s_1) \in \mathscr{C}(\leq)$ and $(t_2,s_2) \in \mathscr{C}(\leq)$ by definition of $\mathscr{C}(\leq)$.
- $t = s = \mathtt{int}$. This case is trivial.

The case $\mathscr{R} = \asymp$ is structurally similar. We omit the details of the proof, just observing that 0 is compatible with every use and that compatibility is closed with respect to $\leq$. □

Theorem C.1 (Theorem 4.1). *If the algorithm returns σ, then σ is a solution for $\mathscr{C}$.*

Proof. Consequence of Lemma C.2. □