Type Reconstruction Algorithms for Deadlock-Free and Lock-Free Linear π-Calculi

Luca Padovani, Tzu-Chun Chen, Andrea Tosatto

▶ **To cite this version:**

Luca Padovani, Tzu-Chun Chen, Andrea Tosatto. Type Reconstruction Algorithms for Deadlock-Free and Lock-Free Linear π-Calculi. 17th International Conference on Coordination Languages and Models (COORDINATION), Jun 2015, Grenoble, France. pp.83-98, 10.1007/978-3-319-19282-6_6 . hal-01774941

HAL Id: hal-01774941

https://inria.hal.science/hal-01774941v1

Submitted on 24 Apr 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Type Reconstruction Algorithms for Deadlock-Free and Lock-Free Linear π-Calculi

Luca Padovani[1], Tzu-Chun Chen[1,2], and Andrea Tosatto[1]

[1] Università di Torino, Italy
[2] Technische Universität Darmstadt, Germany and Purdue University, USA

Abstract. We define complete type reconstruction algorithms for two type systems ensuring deadlock and lock freedom of linear π-calculus processes. Our work automates the verification of deadlock/lock freedom for a non-trivial class of processes that includes interleaved binary sessions and, to great extent, multiparty sessions as well. A Haskell implementation of the algorithms is available.

1 Introduction

Type systems help finding potential errors during the early phases of software development. In the context of communicating processes, typical errors are: making invalid assumptions about the nature of a received message; using a communication channel beyond its nominal capabilities. Some type systems are able to warn against subtler errors, and sometimes can even guarantee liveness properties as well. For instance, the type systems presented in [18] for the linear π-calculus [14] ensure well-typed processes to be deadlock and lock free. Such stronger guarantees come at the cost of a richer type structure, hence of a greater programming effort, when programmers are supposed to explicitly annotate programs with types. In this respect, *type reconstruction* becomes a most wanted tool in the programmer's toolkit: type reconstruction is the procedure that automatically synthesizes, whenever possible, the types of the entities used by a program; in particular, the types of the channels used by a communicating process. In the present work, we describe type reconstruction algorithms for the type systems presented in [18], thereby automating the static deadlock and lock freedom analysis for a non-trivial class of communicating processes.

A *deadlock* is a configuration with pending communications that cannot complete. A paradigmatic example of deadlock modeled in the π-calculus is illustrated below

$$(\nu a,b)\,(\,a?(x).b!x \mid b?(y).a!y\,) \tag{1.1}$$

where the input on a blocks the output on b, and the input on b blocks the output on a. The key idea used in [18] for detecting deadlocks, which is related to earlier works by Kobayashi [11,13], is to associate each channel with a number – called *level* – specifying the relative order in which different channels should be used. In (1.1), this mechanism requires a to have smaller level than b in the left subprocess, and greater level than b in the right one. Since no level assignment can simultaneously satisfy both requirements, (1.1) is flagged as ill typed. This mechanism does not prevent *locks*, namely

configurations where some communication remains pending although the process as a whole can make progress. A deadlock-free configuration that is not lock free is

$$(\nu a)(\ *c?(x).c!x \mid c!a \mid a!42\ ) \tag{1.2}$$

where the communication pending on a cannot complete. There are no interleaved communications on different channels in (1.2), therefore the level-based mechanism spots no apparent issue. The idea put forward in [18] to reject (1.2) is to also associate each channel with another number – called *ticket* – specifying the maximum number of times the channel can travel in a message. With this mechanism in place, (1.2) is ill typed because a would need an infinite number of tickets to travel infinitely many times on c.

Finding appropriate level and tickets for the channels used by a process can be difficult. We remedy to such difficulty with three contributions. First, we develop complete type reconstruction algorithms for the type systems in [18] so that appropriate level and tickets are synthesized automatically, whenever possible. The linear π-calculus [14], for which the type systems are defined, can model a variety of communicating systems with both static and dynamic network topologies. In particular, binary sessions [5] and, to a large extent, also multiparty sessions [18, technical report], can be encoded in it. Second, we purposely use a variant of the linear π-calculus with pairs instead of a polyadic calculus. While this choice has a cost in terms of technical machinery, it allows us to discuss how to deal with structured data types, which are of primary importance in concrete languages but whose integration in linear type systems requires some care [19]. We give evidence that our algorithms scale easily to other data types, including disjoint sums and polymorphic variants. Third, we present the algorithms *assuming* the existence of type reconstruction for the linear π-calculus [9,19]. This approach has two positive upshots: (1) we focus on the aspects of the algorithms concerning deadlock and lock freedom, thereby simplifying their presentation and the formal study of their properties; (2) we show how to combine in a modular way increasingly refined type reconstruction stages and how to address some of the issues that may arise in doing so.

In what follows we review the linear π-calculus with pairs (Section 2) and the type systems for deadlock and lock freedom of [18] (Section 3). Such type systems are unsuitable to be used as the basis for type reconstruction algorithms. So, we reformulate them to obtain reconstruction algorithms that are both correct and complete (Section 4). Then, we sketch an algorithm for solving the constraints generated by the reconstruction algorithms (Section 5) We conclude presenting a few benchmarks, further connections with related work, and directions of future research (Section 6).

The algorithms have been implemented and integrated in a tool for the static analysis of π-calculus processes. The archive with the source code of the tool, available at the page `http://di.unito.it/hypha`, includes a wide range of examples, of which we can discuss only one in the paper because of space constraints.

2 The simply-typed linear π-calculus with pairs

The process language we work with is the asynchronous π-calculus extended in two ways: (1) we generalize names to *expressions* to account for pairs and other data types; (2) we assume that names are explicitly annotated with *simple types* possibly inferred in

a previous reconstruction phase ("simple" means without level/ticket decorations). We annotate free names instead of bound names because, in a behavioral type system, each occurrence of a name may be used according to a different type. Typically, two distinct occurrences of the same linear channel are used for complementary I/O actions. We use $m, n, \ldots$ to range over integer numbers; we use sets of *variables* $x, y, \ldots$ and *channels* $a, b, \ldots$; names $u, v, \ldots$ are either channels or variables; we let *polarities* $p, q, \ldots$ range over subsets of $\{\mathtt{?}, \mathtt{!}\}$; we abbreviate $\{\mathtt{?}\}$ with $\mathtt{?}$, $\{\mathtt{!}\}$ with $\mathtt{!}$, and $\{\mathtt{?}, \mathtt{!}\}$ with $\mathtt{\#}$. *Processes* $P, Q, \ldots$, *expressions* $\mathtt{e}, \mathtt{f}, \ldots$, and *simple types* $t, s, \ldots$ are defined below:

Process	P,Q	::=	$\mathbf{0} \mid \mathtt{e?}(x).P \mid \mathtt{e!f} \mid P \mid Q \mid (\nu a)P \mid *P$
Expression	$\mathtt{e},\mathtt{f}$	::=	$n \mid u^t \mid (\mathtt{e},\mathtt{f}) \mid \mathtt{fst}(\mathtt{e}) \mid \mathtt{snd}(\mathtt{e})$
Simple type	t,s	::=	$\mathtt{int} \mid p[t] \mid p[t]^* \mid t \times s$

Expressions include integer constants, names, pairs, and the two pair projection operators `fst` and `snd`. Simple types are the regular, possibly infinite terms built using the rightmost productions in grammar above and include the type `int` of integers, the type $p[t]$ of linear channels to be used according to the polarity p and carrying messages of type t, the type $p[t]^*$ of unlimited channels to be used according to the polarity p and carrying messages of type t, and the type $t \times s$ of pairs whose components have respectively type t and s. Recall that linear channels are meant to be used for *one* communication, whereas unlimited channels can be used any number of communications. We require every infinite branch of a type to contain infinitely many occurrences of channel constructors. For example, the term t satisfying the equation $t = \mathtt{?}[t]$ is a valid type while the one satisfying the equation $t = t \times \mathtt{int}$ is not. We impose this requirement to simplify the formal development, but it can be lifted (for example, the implementation supports ordinary recursive types such as lists and trees).

Since we are only concerned with type reconstruction, we do not give an operational semantics of the calculus. The interested reader may refer to [14,19] for generic properties of the linear π-calculus and to [18] for the formalization of (dead)lock freedom. We conclude this section with a comprehensive example that is representative of a class of processes for which our type systems are able to prove deadlock and lock freedom.

Example 2.1 (full duplex communication). The term

$$*c\mathtt{?}(x).(\nu a)(\,\mathtt{fst}(x)\mathtt{!}a \;\mid\; \mathtt{snd}(x)\mathtt{?}(y).c\mathtt{!}(a,y)\,) \;\mid\; c\mathtt{!}(e,f) \;\mid\; c\mathtt{!}(f,e)$$

(where we have omitted simple type annotations) models a system composed of two neighbor processes connected by channels e and f. The process spawned by $c\mathtt{!}(e,f)$ uses e for sending a message to the neighbor. Simultaneously, it waits on f for a message from the neighbor. The process spawned by $c\mathtt{!}(f,e)$ does the opposite. Each exchanged message consists of a payload (omitted) and a *continuation channel* on which subsequent messages are exchanged. Above, each process sends and receives a fresh continuation a. Once the two communications have been performed, each process iterates with a new pair of corresponding continuations. ■

3 Type systems for deadlock and lock freedom

In this section we review the type systems ensuring deadlock and lock freedom [18] for which we want to define corresponding reconstruction algorithms. Both type systems

rely on refined linear channel types of the form $p\,[t]^{n}_{m}$ where the decorations n and m are respectively the *level* and the *tickets* of a channel with this type. Intuitively, levels are used for imposing an ordering on the input/output operations performed on channels: channels with lower level must be used *before* channels with higher level; tickets limit the number of "travels" for channels: a channel with m tickets can be sent at most m times in a message. From now on, we use T, S, ... to range over *types*, which have the same structure and constructors as simple types, but where linear channel types are decorated with levels and tickets. We write $\lfloor T \rfloor$ for the *stripping* of T, namely for the simple type obtained by removing all level and ticket decorations from T. For example, $\lfloor \mathtt{?[int} \times \mathtt{![int]}^{n}_{m}\mathtt{]}^{*} \rfloor = \mathtt{?[int} \times \mathtt{![int]]}^{*}$. Note that $\lfloor \cdot \rfloor$ is a non-injective function.

We need some auxiliary operators. First, we extend the notion of level from channel types to arbitrary types. The level of a type T, written $|T|$, is an element of the set $\mathbb{Z} \cup \{\bot, \top\}$ ordered in the obvious way and formally defined thus:

$$|T| \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } T = p\,[S]^{*} \text{ and } \mathtt{?} \in p \\ n & \text{if } T = p\,[S]^{n}_{m} \text{ and } p \neq \emptyset \\ \min\{|T_1|, |T_2|\} & \text{if } T = T_1 \times T_2 \\ \top & \text{otherwise} \end{cases} \tag{3.1}$$

As an example, we have $|\mathtt{int} \times \mathtt{?[![int]}^{1}_{0}\mathtt{]}^{0}_{0}| = \min\{|\mathtt{int}|, |\mathtt{?[![int]}^{1}_{0}\mathtt{]}^{0}_{0}|\} = \min\{\top, 0\} = 0$. Intuitively, the level of T measures the inverse urgency for using values of type T in order to ensure (dead)lock freedom: the lowest level (and highest urgency) $\bot$ is given to unlimited channels with input polarity, for which we want to guarantee input receptiveness; finite levels are reserved for linear channels; the highest level (and lowest urgency) $\top$ is given to values such as numbers or channels with empty polarity whose use is not critical as far as (dead)lock is concerned. Note that $|T|$ is well defined because every infinite branch of T has infinitely many channel constructors.

We also need an operator to *shift* the topmost levels and tickets in types. We define

$$\$^{n}_{m}\,T \stackrel{\text{def}}{=} \begin{cases} p\,[S]^{n+h}_{m+k} & \text{if } T = p\,[S]^{h}_{k} \\ (\$^{n}_{m}\,T_1) \times (\$^{n}_{m}\,T_2) & \text{if } T = T_1 \times T_2 \\ T & \text{otherwise} \end{cases} \tag{3.2}$$

so that, for example, we have $\$^{2}_{1}\,(\mathtt{int} \times \mathtt{?[![int]}^{1}_{0}\mathtt{]}^{0}_{0}) = \mathtt{int} \times \mathtt{?[![int]}^{1}_{0}\mathtt{]}^{2}_{1}$.

Next, we define an operator for *combining* the types of different occurrences of the same object. If an object is used according to type T in one part of a process and according to type S in another part, then it is used according to the type $T + S$ overall, where $T + S$ is inductively defined thus:

$$T + S \stackrel{\text{def}}{=} \begin{cases} \mathtt{int} & \text{if } T = S = \mathtt{int} \\ (T_1 + S_1) \times (T_2 + S_2) & \text{if } T = T_1 \times T_2 \text{ and } S = S_1 \times S_2 \\ (p \cup q)\,[T]^{n}_{h+k} & \text{if } T = p\,[T]^{n}_{h} \text{ and } S = q\,[T]^{n}_{k} \text{ and } p \cap q = \emptyset \\ (p \cup q)\,[T]^{*} & \text{if } T = p\,[T]^{*} \text{ and } S = q\,[T]^{*} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Table 1. Typing rules for the deadlock-free ($k = 0$) and lock-free ($k = 1$) linear π-calculus.

Typing rules for expressions $\boxed{\Gamma \vdash \mathsf{e} : t}$

$$\text{[T-INT]}\ \frac{}{\Gamma \vdash n : \mathtt{int}}\ \mathsf{un}(\Gamma) \qquad \text{[T-NAME]}\ \frac{}{\Gamma, u : T \vdash u^{\lfloor T \rfloor} : T}\ \mathsf{un}(\Gamma)$$

$$\text{[T-PAIR]}\ \frac{\Gamma_i \vdash \mathsf{e}_i : T_i\ ^{(i=1,2)}}{\Gamma_1 + \Gamma_2 \vdash (\mathsf{e}_1, \mathsf{e}_2) : T_1 \times T_2} \qquad \text{[T-FST]}\ \frac{\Gamma \vdash \mathsf{e} : T \times S}{\Gamma \vdash \mathtt{fst}(\mathsf{e}) : T}\ \mathsf{un}(S) \qquad \text{[T-SND]}\ \frac{\Gamma \vdash \mathsf{e} : T \times S}{\Gamma \vdash \mathtt{snd}(\mathsf{e}) : S}\ \mathsf{un}(T)$$

Typing rules for processes $\boxed{\Gamma \vdash_k P}$

$$\text{[T-IN]}\ \frac{\Gamma_1 \vdash \mathsf{e} : \mathtt{?}[T]^{\,n}_{\,m} \qquad \Gamma_2, x : \$^n_0 T \vdash_k P}{\Gamma_1 + \Gamma_2 \vdash_k \mathsf{e}\mathtt{?}(x).P}\ n < |\Gamma_2| \qquad \text{[T-OUT]}\ \frac{\Gamma_1 \vdash \mathsf{e} : \mathtt{!}[T]^{\,n}_{\,m} \qquad \Gamma_2 \vdash \mathsf{f} : \$^n_k T}{\Gamma_1 + \Gamma_2 \vdash_k \mathsf{e}\mathtt{!}\mathsf{f}}\ n < |\Gamma_2|$$

$$\text{[T-IN*]}\ \frac{\Gamma_1 \vdash \mathsf{e} : \mathtt{?}[T]^{\,*} \qquad \Gamma_2, x : T \vdash_k P}{\Gamma_1 + \Gamma_2 \vdash_k *\mathsf{e}\mathtt{?}(x).P}\ \overline{\mathsf{un}}(\Gamma_2) \qquad \text{[T-OUT*]}\ \frac{\Gamma_1 \vdash \mathsf{e} : \mathtt{!}[T]^{\,*} \qquad \Gamma_2 \vdash \mathsf{f} : \$^n_k T}{\Gamma_1 + \Gamma_2 \vdash_k \mathsf{e}\mathtt{!}\mathsf{f}}\ \bot < |\Gamma_2|$$

$$\text{[T-IDLE]}\ \frac{}{\Gamma \vdash_k \mathbf{0}}\ \mathsf{un}(\Gamma) \qquad \text{[T-PAR]}\ \frac{\Gamma_1 \vdash_k P \qquad \Gamma_2 \vdash_k Q}{\Gamma_1 + \Gamma_2 \vdash_k P \mid Q} \qquad \text{[T-NEW]}\ \frac{\Gamma, a : \#[T]^{\,m}_{\,n} \vdash_k P}{\Gamma \vdash_k (\nu a) P} \qquad \text{[T-NEW*]}\ \frac{\Gamma, a : \#[T]^{\,*} \vdash_k P}{\Gamma \vdash_k (\nu a) P}$$

Type combination is partial and is only defined when the combined types have the same structure. In particular, channel types can be combined only if they have equal message types; linear channel types can be combined only if they have disjoint polarities and equal level. Also, the combination of two channel types has the union of their polarities and, in the case of linear channels, the sum of their tickets. For example, a channel that is used both with type $\mathtt{?}[\mathtt{int}]^{\,0}_{\,1}$ and with type $\mathtt{!}[\mathtt{int}]^{\,0}_{\,2}$ is used overall according to the type $\mathtt{?}[\mathtt{int}]^{\,0}_{\,1} + \mathtt{!}[\mathtt{int}]^{\,0}_{\,2} = \#[\mathtt{int}]^{\,0}_{\,3}$.

Lastly, we define *type environments* $\Gamma, \ldots$ as finite maps from names to types written $u_1 : T_1, \ldots, u_n : T_n$. As usual, $\mathsf{dom}(\Gamma)$ is the domain of Γ and Γ_1, Γ_2 is the union of Γ_1 and Γ_2 when $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \emptyset$. We extend type combination to type environments:

$$\Gamma_1 + \Gamma_2 \stackrel{\text{def}}{=} \Gamma_1, \Gamma_2 \qquad \text{if } \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \emptyset$$
$$(\Gamma_1, u : T) + (\Gamma_2, u : S) \stackrel{\text{def}}{=} (\Gamma_1 + \Gamma_2), u : T + S$$

We let $|\Gamma| \stackrel{\text{def}}{=} \min\{|\Gamma(u)| \mid u \in \mathsf{dom}(\Gamma)\}$ be the level of a type environment, we write $\mathsf{un}(\Gamma)$ if $|\Gamma| = \top$ and $\overline{\mathsf{un}}(\Gamma)$ if $\mathsf{un}(\Gamma)$ and Γ has no top-level linear channel types. Note that $\overline{\mathsf{un}}(\Gamma)$ is strictly stronger than $\mathsf{un}(\Gamma)$. For example, if $\Gamma \stackrel{\text{def}}{=} x : \mathtt{int} \times \emptyset[\mathtt{int}]^{\,0}_{\,0}$ we have $\mathsf{un}(\Gamma)$ but not $\overline{\mathsf{un}}(\Gamma)$ because $\Gamma(x)$ has a top-level linear channel type.

The type systems for deadlock and lock freedom are defined by the rules in Table 1 deriving judgments $\Gamma \vdash \mathsf{e} : T$ for expressions and $\Gamma \vdash_k P$ for processes. The type system for deadlock freedom is obtained by taking $k = 0$, whereas the type system for lock

freedom is obtained by taking $k = 1$ and restricting all levels in linear channel types to be non negative. We illustrate the typing rules as we work through the typing derivation of the replicated process in Example 2.1. The interested reader may refer to the implementation or [18] for more examples and detailed descriptions of the rules.

Let T and S be the types defined by the equations $T = \,!\,[S]^0_0 \times ?\,[S]^0_0$ and $S = ?\,[S]^1_1$. We build the derivation bottom up, from the judgment stating that the whole process is well typed. Since the process is a replicated input, we apply [T-IN*] thus:

$$\frac{c : ?\,[T]^* \vdash c : ?\,[T]^* \quad c : !\,[T]^*, x : T \vdash_1 (\nu a)(\mathtt{fst}(x)\,!a \mid \mathtt{snd}(x)?(y).c\,!(a,y))}{c : \#\,[T]^* \vdash_1 *c?(x).(\nu a)(\mathtt{fst}(x)\,!a \mid \mathtt{snd}(x)?(y).c\,!(a,y))}$$

In applying this rule we have $\Gamma_2 = c : !\,[T]^*$ so the side condition $\overline{\mathsf{un}}(\Gamma_2)$ of [T-IN*] is satisfied: since a replicated input process is permanently available, its body cannot contain any free linear channel except those possibly received through the unlimited channel. The side condition $\overline{\mathsf{un}}(\Gamma_2)$, which is stronger than simply $\mathsf{un}(\Gamma_2)$, makes sure that a replicated input process does not contain linear channels and therefore is level polymorphic. We will see a use of this feature at the very end of the derivation. The continuation of the process gains visibility of the message x with type T and is a restriction of a linear channel a. Hence, the next step is an obvious application of [T-NEW]:

$$\frac{c : !\,[T]^*, x : T, a : \#\,[S]^1_3 \vdash_1 \mathtt{fst}(x)\,!a \mid \mathtt{snd}(x)?(y).c\,!(a,y)}{c : !\,[T]^*, x : T \vdash_1 (\nu a)(\mathtt{fst}(x)\,!a \mid \mathtt{snd}(x)?(y).c\,!(a,y))} \tag{3.3}$$

We guess level 1 and 3 tickets for a. The rationale is that a is a continuation channel that will be used *after* the channels in x, which have level 0, so a must have strictly positive level. Also, in Example 2.1 the channel a travels three times. At this point the typing derivation forks, for we deal with the parallel composition of two processes. This means that we have to split the type environment in two parts, each describing the resources used by the corresponding subprocess in (3.3). We have $\Gamma = \Gamma_1 + \Gamma_2$ where

$$\Gamma \stackrel{\text{def}}{=} c : !\,[T]^*, x : T, a : \#\,[S]^1_3 \qquad \begin{array}{ll} \Gamma_1 \stackrel{\text{def}}{=} & x : !\,[S]^0_0 \times \emptyset\,[S]^0_0,\ a : ?\,[S]^1_2 \\ \Gamma_2 \stackrel{\text{def}}{=} & c : !\,[T]^*,\ x : \emptyset\,[S]^0_0 \times ?\,[S]^0_0,\ a : !\,[S]^1_1 \end{array}$$

Observe that Γ is split in such a way that: c only occurs in Γ_2, because it is only used in the right subprocess in (3.3); in each subprocess, the unused linear channel in the pair x is given empty polarity; the type of the continuation a has input polarity (and 2 tickets) in Γ_1 and output polarity (and 1 ticket) in Γ_2. The type of a in Γ_1 is the same as $\$^0_1 S$, and we use the latter form from now on. We complete the typing derivation for the left subprocess in (3.3) using Γ_1 and applying [T-OUT]:

$$\frac{x : !\,[S]^0_0 \times \emptyset\,[S]^0_0 \vdash \mathtt{fst}(x) : !\,[S]^0_0 \qquad a : \$^0_1 S \vdash a : \$^0_1 S}{x : !\,[S]^0_0 \times \emptyset\,[S]^0_0, a : \$^0_1 S \vdash_1 \mathtt{fst}(x)\,!a}\ 0 < |\$^0_1 S| = 1$$

The side condition $0 < 1$ ensures that the message has higher level than the channel on which it travels, according to the intuition that the message can only be used *after* the communication has occurred and the message has been received. In this case, the level of $\mathtt{fst}(x)$ is 0 that is smaller than the level of a, which is 1. Shifting the tickets

from the type of a consumes one of its tickets, meaning that after this communication a gets closer to the point where it must be the subject of a communication.

Concerning the right subprocess in (3.3), we use Γ_2 above and apply [T-IN] to obtain

$$\frac{x : \emptyset[S]^{\,0}_{\,0} \times ?[S]^{\,0}_{\,0} \vdash \mathtt{snd}(x) : ?[S]^{\,0}_{\,0} \qquad c : ![T]^*, a : ![S]^{\,1}_{\,1}, y : S \vdash_1 c!(a,y)}{c : ![T]^*, x : \emptyset[S]^{\,0}_{\,0} \times ?[S]^{\,0}_{\,0}, a : ![S]^{\,1}_{\,1} \vdash_1 \mathtt{snd}(x)?(y).c!(a,y)}\ 0 < 1$$

The side condition $0 < 1$ checks that the level of the linear channel used for input is smaller than the level of any other channel occurring free in the continuation of the process. In this case, c has level $\top$ because it is an unlimited channel with output polarity, whereas a has level 1. To close the derivation we must type the recursive invocation of c. We do so with an application of [T-OUT*]:

$$\frac{c : ![T]^* \vdash c : ![T]^* \qquad a : ![S]^{\,1}_{\,1}, y : S \vdash (a,y) : \$^{1}_{1} T}{c : ![T]^*, a : ![S]^{\,1}_{\,1}, y : S \vdash_1 c!(a,y)}\ \bot < 1$$

The side condition $\bot < 1$ ensures that no unlimited channel with input polarity is sent in the message. This is necessary to guarantee input receptiveness on unlimited channels. There is a mismatch between the actual type $\$^{1}_{1} T$ and the expected type T of the message. The shifting on the tickets is due, once again, to the fact that 1 ticket is required and consumed for the channels to travel. The shifting on the levels realizes a form of *level polymorphism* whereby we are allowed to send on c a pair of channels with level 1 even if c expects a pair of channels with level 0. This is safe because we know, from the side condition of [T-IN*], that the receiver of the message does not own any linear channel except those possibly contained in the message itself. Therefore, the exact level of the channels in the message is irrelevant, as long as it is obtained by shifting of the expected message type. Level polymorphism is a key distinguishing feature of our type systems that makes it possible to deal with non-trivial recursive processes.

4 Type reconstruction

We now face the problem of defining a type reconstruction algorithm for the type system presented in the previous section. The input of the algorithm is a process P where names are explicitly annotated with simple types, possibly resulting from a previous reconstruction stage [9,19]. Notwithstanding such explicit annotations, the typing rules in Table 1 rely on guesses concerning (i) the splitting of type environments, (ii) levels and tickets that decorate linear channel types, and (iii) how tickets are distributed in combined types. We address these issues using standard strategies. Concerning (i), we synthesize type environments for expressions and processes by looking at the free names occurring in them. Concerning (ii) and (iii), we proceed in two steps: first, we transform each simple type t in P into a *type expression* T that has the same structure as t, but where we use fresh level and ticket *variables* in every slot where a level or a ticket is expected; we call this transformation *dressing*. Then, we accumulate (rather than check) the constraints that these level and ticket variables should satisfy, as by the side conditions of the typing rules (Table 1). Finally, we look for a solution of these

constraints. It turns out that the accumulated constraints can always be expressed as an integer programming problem for which there exist dedicated solvers.

There is still a subtle source of ambiguity in the procedure outlined so far. We have remarked that stripping is a non-injective function, meaning that different types may be stripped to the same simple type. For example, if we take $T = ?[T]^{1}_{1}$ and $S = ?[T]^{0}_{0}$ we have $\lfloor T \rfloor = \lfloor S \rfloor = s$ where $s = ?[s]$. Now, if we were to reconstruct either T or S from s, we would have to dress s with level and ticket variables in every slot where a level or a ticket is expected. But since s is infinite, such dressing is not unique. For example, $\mathsf{T} = ?[\mathsf{T}]^{\eta_1}_{\theta_1}$ and $\mathsf{S} = ?[\mathsf{T}]^{\eta_2}_{\theta_2}$ are just two of the infinitely many possible dressings of s with level and ticket variables: in T we have used two distinct variables η_1 and θ_1, one for each slot; in S we have used four. The problem is that from the dressing T we can only reconstruct T, by taking $\eta_1 = \theta_1 = 1$, whereas from the dressing S we can reconstruct both T (by assigning all variables to 1) as well as S, by taking $\eta_1 = \theta_1 = 1$ and $\eta_2 = \theta_2 = 0$. This means that the choice of the number of integer variables we use in dressing (infinite) simple types constrains the types that we can reconstruct from them, which is a risk for the completeness of the type reconstruction algorithms. To cope with this issue, we dress simple types *lazily*, only to their topmost linear channel constructors, and we put fresh type variables in place of message types, leaving them undressed. It is only when the message is used that we (lazily) dress its type as well. The introduction of fresh type variables for message types means that we redo part of the work already carried out for reconstructing simple types [19]. This appears to be an inevitable price to pay to have completeness of the type reconstruction algorithms, when they build on top of (instead of being performed together with) previous stages.

To formalize the algorithms, we introduce countable sets of *type variables* α, β and of *integer variables* η, θ; *type expressions* and *integer expressions* are defined below:

$$\begin{array}{rrcl} \textbf{Type expression} & \mathsf{T}, \mathsf{S} & ::= & \texttt{int} \mid \alpha \mid p[\mathsf{T}]^{\lambda}_{\tau} \mid p[\mathsf{T}]^{*} \mid \mathsf{T} \times \mathsf{S} \\ \textbf{Integer expression} & \lambda, \varepsilon, \tau & ::= & n \mid \eta \mid \varepsilon + \varepsilon \mid \varepsilon - \varepsilon \end{array}$$

Type expressions differ from types in three ways: they are always finite, they have integer expressions in place of levels and tickets, and they include type variables α denoting unknown types awaiting to be lazily dressed. Integer expressions are linear polynomials of integer variables.

We say that T is *proper*, written $\mathsf{prop}(\mathsf{T})$, if all the type variables in T are guarded by a channel constructor. For example, both $\texttt{int}$ and $p[\alpha]^{*}$ are proper (all type variables occur within channel types), but α and $\texttt{int} \times \alpha$ are not. Since the level and tickets of a type expression are solely determined by its top-level linear channel constructors, properness characterizes those type expressions that are "sufficiently dressed" so that it is possible to extract their level and to combine them with other type expressions, even if these type expressions contain type variables.

We now revisit and adapt all the auxiliary operators and notions defined for types to type expressions. Recall that the level of T is the minimum level of any topmost linear channel type in T, or $\bot$ if T has a topmost unlimited channel type with input polarity. Since a type expression T may contain unevaluated level expressions, we cannot compute a minimum level in general. However, a quick inspection of Table 1 reveals that minima of levels always occur on the right hand side of inequalities, and an inequality like $n < \min\{m_i \mid i \in I\}$ can equivalently be expressed as a set of inequalities

$\{n < m_i \mid i \in I\}$. Following this observation, we define the *level* $|\mathsf{T}|$ of a proper type expression T as the *set* of level expressions that decorate the topmost linear channel types in T, and possibly the element $\bot$. Formally:

$$|\mathsf{T}| \stackrel{\text{def}}{=} \begin{cases} \{\bot\} & \text{if } \mathsf{T} = p\,[\mathsf{S}]^* \text{ and } ? \in p \\ \{\lambda\} & \text{if } \mathsf{T} = p\,[S]^{\lambda}_{\tau} \text{ and } p \neq \emptyset \\ |\mathsf{T}_1| \cup |\mathsf{T}_2| & \text{if } \mathsf{T} = \mathsf{T}_1 \times \mathsf{T}_2 \\ \emptyset & \text{otherwise} \end{cases} \tag{4.1}$$

We write $\mathsf{un}(\mathsf{T})$ if $|\mathsf{T}| = \emptyset$, in which case T denotes an unlimited type. *Shifting* for proper type expressions is defined just like for types, except that we symbolically record the sum of level/ticket expressions instead of computing it:

$$\$^{\lambda}_{\tau}\,\mathsf{T} \stackrel{\text{def}}{=} \begin{cases} p\,[\mathsf{S}]^{\lambda+\lambda'}_{\tau+\tau'} & \text{if } \mathsf{T} = p\,[\mathsf{S}]^{\lambda'}_{\tau'} \\ (\$^{\lambda}_{\tau}\,\mathsf{T}_1) \times (\$^{\lambda}_{\tau}\,\mathsf{T}_2) & \text{if } \mathsf{T} = \mathsf{T}_1 \times \mathsf{T}_2 \\ \mathsf{T} & \text{otherwise} \end{cases} \tag{4.2}$$

Because type expressions may contain type and integer variables, we cannot determine *a priori* whether the combination of two type expressions is possible. For instance, the combination of $?\,[\mathsf{T}]^{\lambda_1}_{\tau_1}$ and $!\,[\mathsf{S}]^{\lambda_2}_{\tau_2}$ is possible only if T and S denote the same type and if λ_1 and λ_2 evaluate to the same level. We cannot check these conditions right away, when T, S and the level expressions contain variables. Instead, we record these conditions into a *constraint*. Constraints φ, … are conjunctions of *type constraints* $\mathsf{T} = \mathsf{S}$ (equality relations between type expressions) and *integer constraints* $\varepsilon \leq \varepsilon'$ (inequality relations between integer expressions). Formally, their syntax is defined by

$$\textbf{Constraint} \qquad \varphi \quad ::= \quad \mathtt{true} \;\mid\; \mathsf{T} = \mathsf{T} \;\mid\; \varepsilon \leq \varepsilon \;\mid\; \varphi \wedge \varphi$$

We write $\varepsilon < \varepsilon'$ in place of $\varepsilon + 1 \leq \varepsilon'$ and $\varepsilon = \varepsilon'$ in place of $\varepsilon \leq \varepsilon' \wedge \varepsilon' \leq \varepsilon$; if $\mathscr{E} = \{\varepsilon_i\}_{i \in I}$ is a finite set of integer expressions, we write $\varepsilon < \mathscr{E}$ for the constraint $\bigwedge_{i \in I} \varepsilon < \varepsilon_i$; finally, we write $\mathsf{dom}(\varphi)$ for the set of type expressions occurring in φ.

The *combination* operator $\mathsf{T} \sqcup \mathsf{S}$ for type expressions returns a pair $\mathsf{R}; \varphi$ made of the resulting type expression R and the constraint φ that must be satisfied for the combination to be possible. The definition of $\sqcup$ mimics exactly that of $+$ in Section 3, except that all non-checkable conditions accumulate in constraints:

$$\mathsf{T} \sqcup \mathsf{S} \stackrel{\text{def}}{=} \begin{cases} \mathtt{int} & ; \mathtt{true} & \text{if } \mathsf{T} = \mathtt{int} \text{ and } \mathsf{S} = \mathtt{int} \\ (p \cup q)\,[\mathsf{T}']^{\lambda}_{\tau+\tau'} & ; \mathsf{T}' = \mathsf{S}' \wedge \lambda = \lambda' & \text{if } \mathsf{T} = p\,[\mathsf{T}']^{\lambda}_{\tau} \text{ and } \mathsf{S} = q\,[\mathsf{S}']^{\lambda'}_{\tau'} \\ & & \quad \text{and } p \cap q = \emptyset \\ (p \cup q)\,[\mathsf{T}']^* & ; \mathsf{T}' = \mathsf{S}' & \text{if } \mathsf{T} = p\,[\mathsf{T}']^* \text{ and } \mathsf{S} = q\,[\mathsf{S}']^* \\ \mathsf{R}_1 \times \mathsf{R}_2 & ; \varphi_1 \wedge \varphi_2 & \text{if } \mathsf{T} = \mathsf{T}_1 \times \mathsf{T}_2 \text{ and } \mathsf{S} = \mathsf{S}_1 \times \mathsf{S}_2 \\ & & \quad \text{and } \mathsf{T}_i \sqcup \mathsf{S}_i = \mathsf{R}_i; \varphi_i \\ \text{undefined} & & \text{otherwise} \end{cases}$$

Like type combination, also $\sqcup$ is a partial operator: $\mathsf{T} \sqcup \mathsf{S}$ is undefined if T and S are structurally incompatible (*e.g.*, if $\mathsf{T} = \mathtt{int}$ and $\mathsf{S} = p\,[\mathtt{int}]^*$) or if T and S are not proper. When $\mathsf{T} \sqcup \mathsf{S}$ is defined, though, the resulting type expression is always proper.

We use $\Delta, \ldots$ to range over *type expression environments* (or just environments, for short), namely finite maps from names to type expressions, and we inherit all the notation introduced for type environments. We let $|\Delta| \stackrel{\text{def}}{=} \bigcup_{u \in \mathsf{dom}(\Delta)} |\Delta(u)|$ and write $\overline{\mathsf{un}}(\Delta)$ if $|\Delta| = \emptyset$ and Δ has no top-level linear channel type in its range. By now, the extension of $\sqcup$ to environments is easy to imagine: when defined, $\Delta_1 \sqcup \Delta_2$ is a pair $\Delta; \varphi$ made of the resulting environment Δ and of a constraint φ that results from the combination of the type expressions in Δ_1 and Δ_2. More precisely:

$$\begin{array}{rll} \Delta_1 \sqcup \Delta_2 \stackrel{\text{def}}{=} \Delta_1, \Delta_2 \ ; \mathsf{true} & & \text{if } \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2) = \emptyset \\ (\Delta_1, u : \mathsf{T}) \sqcup (\Delta_2, u : \mathsf{S}) \stackrel{\text{def}}{=} \Delta, u : \mathsf{R} \ ; \varphi \wedge \varphi' & & \text{if } \Delta_1 \sqcup \Delta_2 = \Delta; \varphi \text{ and } \mathsf{T} \sqcup \mathsf{S} = \mathsf{R}; \varphi' \end{array}$$

The last notion we need to formalize, before introducing the reconstruction algorithms, is that of *dressing*. Dressing a simple type t means placing fresh integer variables in the level/ticket slots of t. Formally, we say that T is a *dressing* of t if $t \uparrow \mathsf{T}$ is inductively derivable by the following rules which pick globally fresh variables:

$$\mathtt{int} \uparrow \mathtt{int} \qquad \frac{\alpha \text{ fresh}}{p\,[t]^* \uparrow p\,[\alpha]^*} \qquad \frac{\alpha, \eta, \theta \text{ fresh}}{p\,[t] \uparrow p\,[\alpha]^{\eta}_{\theta}} \qquad \frac{t_i \uparrow \mathsf{T}_i \ ^{(i=1,2)}}{t_1 \times t_2 \uparrow \mathsf{T}_1 \times \mathsf{T}_2}$$

Note that the decoration of t with fresh integer variables stops at the topmost channel types in t and that message types are left undecorated. By definition, the dressing of a simple type is always a proper type expression.

We can now present the type reconstruction algorithms, defined by the rules in Table 2. The rules in the upper part of the table derive judgments of the form $\mathsf{e} : \mathsf{T} \blacktriangleright \Delta; \varphi$, stating that e has type T in the environment Δ if the constraint φ is satisfied. The expression e is the only "input" of the judgment, while T, Δ, and φ are synthesized from it. There is a close correspondence between these rules and those for expressions in Table 1. Observe the use of $\sqcup$ where $+$ was used in Table 1, the accumulation of constraints from the premises to the conclusion of each rule and, most notably, the dressing of the simple type that annotates u in [I-NAME]. Type expressions synthesized by the rules are always proper, so the side conditions in [I-FST] and [I-SND] can be safely checked.

The rules in the lower part of the table derive judgments of the form $P \blacktriangleright_k \Delta; \varphi$, stating that P is well typed in the environment Δ if the constraint φ is satisfied. The parameter k plays the same role as in the type system (Table 1). The process P and the parameter k are the only "inputs" of the judgments, and Δ and φ are synthesized from them. All rules except [I-WEAK] have a corresponding one in Table 1. Like for expressions, environments are combined through $\sqcup$ and constraints accumulate from premises to conclusions. We focus on the differences with respect to the typing rules.

In rule [T-IN], the side condition verifies that the level of the channel e on which an input is performed is smaller than the level of any channel used for typing the continuation process P. This condition can be decomposed in two parts: (1) no unlimited channel with input polarity must be in P; this condition is necessary to ensure input receptiveness on unlimited channels in the original type system [18] and is expressed in [I-IN] as the side condition $\bot \notin |\Delta_2|$, which can be checked on type expressions environments directly; (2) the level of e must satisfy the ordering with respect to all the linear channels in P; this is expressed in [I-IN] as the constraint $\lambda < |\Delta_2|$, where λ is the level of e. The same side condition and constraint are found in [I-OUT].

Table 2. Type reconstruction rules for expressions and processes.

Reconstruction rules for expressions $\boxed{\mathsf{e} : \mathsf{T} \blacktriangleright \Delta; \varphi}$

$$\text{[I-INT]}\ \frac{}{n : \mathtt{int} \blacktriangleright \emptyset; \mathtt{true}} \qquad \text{[I-PAIR]}\ \frac{\mathsf{e}_i : \mathsf{T}_i \blacktriangleright \Delta_i; \varphi_i \ ^{(i=1,2)}}{(\mathsf{e}_1, \mathsf{e}_2) : \mathsf{T}_1 \times \mathsf{T}_2 \blacktriangleright \Delta; \bigwedge_{1 \le i \le 3} \varphi_i}\ \Delta_1 \sqcup \Delta_2 = \Delta; \varphi_3$$

$$\text{[I-NAME]}\ \frac{}{u^t : \mathsf{T} \blacktriangleright u : \mathsf{T}; \mathtt{true}}\ t \uparrow \mathsf{T} \qquad \text{[I-FST]}\ \frac{\mathsf{e} : \mathsf{T} \times \mathsf{S} \blacktriangleright \Delta; \varphi}{\mathtt{fst}(\mathsf{e}) : \mathsf{T} \blacktriangleright \Delta; \varphi}\ \mathsf{un}(\mathsf{S}) \qquad \text{[I-SND]}\ \frac{\mathsf{e} : \mathsf{T} \times \mathsf{S} \blacktriangleright \Delta; \varphi}{\mathtt{snd}(\mathsf{e}) : \mathsf{S} \blacktriangleright \Delta; \varphi}\ \mathsf{un}(\mathsf{T})$$

Reconstruction rules for processes $\boxed{P \blacktriangleright_k \Delta; \varphi}$

$$\text{[I-WEAK]}\ \frac{P \blacktriangleright_k \Delta; \varphi}{P \blacktriangleright_k \Delta, u : \mathsf{T}; \varphi}\ \begin{array}{l}\mathsf{un}(\mathsf{T}) \\ \mathsf{prop}(\mathsf{T})\end{array} \qquad \text{[I-IDLE]}\ \frac{}{\mathbf{0} \blacktriangleright_k \emptyset; \mathtt{true}} \qquad \text{[I-PAR]}\ \frac{P_i \blacktriangleright_k \Delta_i; \varphi_i \ ^{(i=1,2)}}{P_1 \mid P_2 \blacktriangleright_k \Delta; \bigwedge_{1 \le i \le 3} \varphi_i}\ \Delta_1 \sqcup \Delta_2 = \Delta; \varphi_3$$

$$\text{[I-IN]}\ \frac{\mathsf{e} : ?[\mathsf{T}]^{\lambda}_{\tau} \blacktriangleright \Delta_1; \varphi_1 \qquad P \blacktriangleright_k \Delta_2, x : \mathsf{S}; \varphi_2}{\mathsf{e}?(x).P \blacktriangleright_k \Delta; \bigwedge_{1 \le i \le 3} \varphi_i \wedge \mathsf{T} = \$_0^{-\lambda}\, \mathsf{S} \wedge \lambda < |\Delta_2|}\ \begin{array}{l}\bot \notin |\Delta_2| \\ \Delta_1 \sqcup \Delta_2 = \Delta; \varphi_3\end{array}$$

$$\text{[I-OUT]}\ \frac{\mathsf{e} : ![\mathsf{T}]^{\lambda}_{\tau} \blacktriangleright \Delta_1; \varphi_1 \qquad \mathsf{f} : \mathsf{S} \blacktriangleright \Delta_2; \varphi_2}{\mathsf{e}!\mathsf{f} \blacktriangleright_k \Delta; \bigwedge_{1 \le i \le 3} \varphi_i \wedge \mathsf{T} = \$_{-k}^{-\lambda}\, \mathsf{S} \wedge \lambda < |\Delta_2|}\ \begin{array}{l}\bot \notin |\Delta_2| \\ \Delta_1 \sqcup \Delta_2 = \Delta; \varphi_3\end{array} \qquad \text{[I-NEW]}\ \frac{P \blacktriangleright_k \Delta, a : \#[\mathsf{T}]^{\lambda}_{\tau}; \varphi}{(\nu a) P \blacktriangleright_k \Delta; \varphi}$$

$$\text{[I-IN*]}\ \frac{\mathsf{e} : ?[\mathsf{T}]^{*} \blacktriangleright \Delta_1; \varphi_1 \qquad P \blacktriangleright_k \Delta_2, x : \mathsf{S}; \varphi_2}{*\mathsf{e}?(x).P \blacktriangleright_k \Delta; \bigwedge_{1 \le i \le 3} \varphi_i \wedge \mathsf{T} = \mathsf{S}}\ \begin{array}{l}\overline{\mathsf{un}}(\Delta_2) \\ \Delta_1 \sqcup \Delta_2 = \Delta; \varphi_3\end{array}$$

$$\text{[I-OUT*]}\ \frac{\mathsf{e} : ![\mathsf{T}]^{*} \blacktriangleright \Delta_1; \varphi_1 \qquad \mathsf{f} : \mathsf{S} \blacktriangleright \Delta_2; \varphi_2}{\mathsf{e}!\mathsf{f} \blacktriangleright_k \Delta; \bigwedge_{1 \le i \le 3} \varphi_i \wedge \mathsf{T} = \$_{-k}^{-\eta}\, \mathsf{S}}\ \begin{array}{l}\bot \notin |\Delta_2| \\ \Delta_1 \sqcup \Delta_2 = \Delta; \varphi_3 \\ \eta \text{ fresh}\end{array} \qquad \text{[I-NEW*]}\ \frac{P \blacktriangleright_k \Delta, a : \#[\mathsf{T}]^{*}; \varphi}{(\nu a) P \blacktriangleright_k \Delta; \varphi}$$

In [T-IN], [T-OUT], and [T-OUT*], shifting is used for updating message levels, consuming tickets, and realizing level polymorphism. In rules [I-IN], [I-OUT], and [I-OUT*], analogous shiftings are performed on type expressions, except that they are inverted and recorded in constraints. For example, when typing the continuation P of a process $\mathsf{e}?(x).P$ using [T-IN], if e has type $?[T]^{n}_{m}$ then the type of x is required to be $\$_0^n T$. In the reconstruction algorithm, we record this requirement as the constraint $\mathsf{T} = \$_0^{-\lambda}\, \mathsf{S}$, where S is the type synthesized for x in P. We invert the shifting because shifting is defined only on proper type expressions, and in [I-IN] (and the other rules mentioned) only S is guaranteed to be proper, while T in general is not.

Finally, note that [I-WEAK] has no correspondent rule in Table 1. This rule is necessary because the premises of [I-IN], [I-IN*], [I-NEW], and [I-NEW*] *assume* that bound names occur in their scope. Since type environments are generated by the algorithm as it works

through an expression or a process, this assumption may not hold if a bound name is never used in its scope. Naturally, the type T of an unused name must be unlimited, whence the constraint $\mathsf{un}(\mathsf{T})$. We also require T to be proper, to preserve the invariant that all environments synthesized by the algorithms have proper types. In principle, [I-WEAK] makes the rule set in Table 2 not syntax directed, which 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.

To state the properties of the reconstruction algorithm, we need a notion of constraint satisfiability. A *variable assignment* σ is a map from type/integer variables to types/integers. We say that σ *covers* X if σ provides assignments to all the type/integer variables occurring in X, where X may be a constraint, a type/integer expression, or an environment. When σ covers X, the *application* of σ to X, written $\sigma\mathsf{X}$, substitutes all type/integer variables according to σ and evaluates all integer expressions in X. When σ covers φ, we say that σ *satisfies* φ if $\sigma \vDash \varphi$ is derivable by the rules:

$$\frac{}{\sigma \vDash \mathsf{true}} \qquad \frac{}{\sigma \vDash \mathsf{T} = \mathsf{S}}\ \sigma\mathsf{T} = \sigma\mathsf{S} \qquad \frac{}{\sigma \vDash \varepsilon \leq \varepsilon'}\ \sigma\varepsilon \leq \sigma\varepsilon' \qquad \frac{\sigma \vDash \varphi_i \ ^{(i=1,2)}}{\sigma \vDash \varphi_1 \wedge \varphi_2}$$

Whenever we apply an assignment σ to a set of type expressions in reference to a derivation that is parametric on k, we will implicitly assume that all integer expressions in ticket slots evaluate to non-negative integers and that, if $k = 1$, all integer expressions in level slots evaluate to non-negative integers. The value of k and the set of type expressions will always be clear from the context.

The reconstruction algorithm is *correct*, namely each derivation obtained through the algorithm such that the resulting constraint is satisfiable corresponds to a derivation in the type system:

Theorem 4.1 (correctness). *If $P \blacktriangleright_k \Delta; \varphi$ and $\sigma \vDash \varphi$ and σ covers Δ, then $\sigma\Delta \vdash_k P$.*

The algorithm is also *complete*, meaning that if there exists a typing derivation for the judgment $\Gamma \vdash_k P$, then the algorithm is capable of synthesizing an environment Δ from which Γ can be obtained by means of a suitable variable assignment:

Theorem 4.2 (completeness). *If $\Gamma \vdash_k P$, then $P \blacktriangleright_k \Delta; \varphi$ for some Δ, φ, and σ such that $\sigma \vDash \varphi$ and $\Gamma = \sigma\Delta$.*

Note that the above results do not give any information about how to verify whether there exists a σ such that $\sigma \vDash \varphi$ and, in this case, how to find such σ. These problems will be addressed in Section 5. We conclude this section showing the reconstruction algorithm at work on the replicated process in Example 2.1.

Example 4.1. Below is the replicated process in Example 2.1, where we have numbered and named the relevant rules used by the algorithm as it visits the process bottom-up, left-to-right:

$$\underbrace{*c?(x)}_{(5)\ [\text{I-IN}*]}.(\nu a)(\overbrace{\underbrace{\mathtt{fst}(x)!a}_{(1)\ [\text{I-OUT}]} \mid \underbrace{\mathtt{snd}(x)?(y)}_{(3)\ [\text{I-IN}]}.\underbrace{c!(a,y)}_{(2)\ [\text{I-OUT}*]}}^{(4)\ [\text{I-PAR}]})$$

Table 3. Type environment and constraints generated for the process in Example 2.1.

i	c	x	a	y	**Constraint**
(1)		$!\,[\alpha_1]^{\eta_1}_{\theta_1} \times \emptyset\,[\alpha_2]^{\eta_2}_{\theta_2}$	$?\,[\alpha_3]^{\eta_3}_{\theta_3}$		$\alpha_1 = ?\,[\alpha_3]^{\eta_3-\eta_1}_{\theta_3-k} \wedge \eta_1 < \eta_3$
(2)	$!\,[\alpha_4]^*$		$!\,[\alpha_5]^{\eta_5}_{\theta_5}$	$?\,[\alpha_6]^{\eta_6}_{\theta_6}$	$\alpha_4 = !\,[\alpha_5]^{\eta_5-\eta_4}_{\theta_5-k} \times ?\,[\alpha_6]^{\eta_6-\eta_4}_{\theta_6-k}$
(3)		$\emptyset\,[\alpha_7]^{\eta_7}_{\theta_7} \times ?\,[\alpha_8]^{\eta_8}_{\theta_8}$			$\alpha_8 = ?\,[\alpha_6]^{\eta_6-\eta_8}_{\theta_6} \wedge \eta_8 < \eta_5$
(4)		$!\,[\alpha_1]^{\eta_1}_{\theta_1+\theta_7} \times ?\,[\alpha_2]^{\eta_2}_{\theta_2+\theta_8}$	$\#\,[\alpha_3]^{\eta_3}_{\theta_3+\theta_5}$		$\alpha_1 = \alpha_7 \wedge \alpha_2 = \alpha_8 \wedge \alpha_3 = \alpha_5$ $\wedge\, \eta_1 = \eta_7 \wedge \eta_2 = \eta_8 \wedge \eta_3 = \eta_5$
(5)	$\#\,[\alpha_9]^*$				$\alpha_9 = !\,[\alpha_1]^{\eta_1}_{\theta_1+\theta_7} \times ?\,[\alpha_2]^{\eta_2}_{\theta_2+\theta_8}$ $\wedge\, \alpha_9 = \alpha_4$

Table 4. Constraint entailment rules.

$$\frac{}{\varphi \vdash_1 0 \le \lambda}\ p\,[\mathsf{T}]^{\lambda}_{\tau} \in \mathsf{dom}(\varphi)\ \text{[S-LEVEL]} \qquad \frac{}{\varphi \vdash_k 0 \le \tau}\ p\,[\mathsf{T}]^{\lambda}_{\tau} \in \mathsf{dom}(\varphi)\ \text{[S-TICKET]}$$

$$\frac{}{\varphi_1 \wedge \varphi_2 \vdash_k \varphi_i}\ i \in \{1,2\}\ \text{[S-CONJ]} \qquad \frac{\varphi \vdash_k \mathsf{T} = \mathsf{S}}{\varphi \vdash_k \mathsf{S} = \mathsf{T}}\ \text{[S-SYMM]} \qquad \frac{\varphi \vdash_k \mathsf{T} = \mathsf{R} \qquad \varphi \vdash_k \mathsf{R} = \mathsf{S}}{\varphi \vdash_k \mathsf{T} = \mathsf{S}}\ \text{[S-TRANS]}$$

$$\frac{\varphi \vdash_k p\,[\mathsf{T}]^{\lambda_1}_{\tau_1} = p\,[\mathsf{S}]^{\lambda_2}_{\tau_2}}{\varphi \vdash_k \mathsf{T} = \mathsf{S} \wedge \lambda_1 = \lambda_2 \wedge \tau_1 = \tau_2}\ \text{[S-CHAN]} \qquad \frac{\varphi \vdash_k p\,[\mathsf{T}]^* = p\,[\mathsf{S}]^*}{\varphi \vdash_k \mathsf{T} = \mathsf{S}}\ \text{[S-CHAN*]} \qquad \frac{\varphi \vdash_k \mathsf{T}_1 \times \mathsf{T}_2 = \mathsf{S}_1 \times \mathsf{S}_2}{\varphi \vdash_k \mathsf{T}_1 = \mathsf{S}_1 \wedge \mathsf{T}_2 = \mathsf{S}_2}\ \text{[S-PAIR]}$$

Each subprocess triggers one rule of the reconstruction algorithm which synthesizes a type environment and possibly generates some constraints. Table 3 summarizes the parts of the environments and the constraints produced at each step of the reconstruction algorithm with parameter k. We have omitted the step concerning the restriction on a, which just removes a from the environment and introduces no constraints. ■

5 Constraint solving

We sketch an algorithm that determines whether a constraint φ is satisfiable and, in this case, computes an assignment that satisfies it. The presentation is somewhat less formal since the key steps of the algorithm are instances of well-known techniques. The algorithm is structured in three phases, *saturation*, *verification*, and *synthesis*.

The constraint φ produced by the reconstruction algorithm does not necessarily mention all the relations that must hold between integer variables. For example, the constraint $\eta_3 - \eta_1 = \eta_6 - \eta_8 \wedge \theta_3 - k = \theta_6$ is implied by those in Table 3, but it appears nowhere. Finding all the integer constraints entailed by a given φ, regardless of whether such constraints are implicit or explicit, is essential because we use an external solver for solving them. The aim of the *saturation phase* is to find all such integer constraints. Table 4 defines an inference system for deriving entailments $\varphi \vdash_k \varphi'$. The parameter k plays the same role as in the type system. Rules [S-LEVEL] and [S-TICKET] introduce non-negativity constraints for integer expressions that occur in level and ticket slots; level

expressions are required to be non-negative only for lock freedom analysis, when $k = 1$; rule [S-CONJ] decomposes conjunctions; rules [S-SYMM] and [S-TRANS] compute the symmetric and transitive closure of type equality; finally, [S-CHAN], [S-CHAN*], and [S-PAIR] state expected congruence rules. We let $\widehat{\varphi} \stackrel{\text{def}}{=} \bigwedge_{\varphi \vdash_k \varphi'} \varphi'$. Clearly $\widehat{\varphi}$ can be computed in finite time and is satisfiable by the same assignments as (*i.e.*, it is equivalent to) φ.

The *verification phase* checks whether $\widehat{\varphi}$ is satisfiable and, in this case, computes an assignment σ_{int} that satisfies the integer constraints in it. In $\widehat{\varphi}$ all the integer constraints are explicit. These are typical constraints of an integer programming problem, for which it is possible to use dedicated (complete) solvers that find a σ_{int} when it exists (our tool supports `GLPK`[3] and `lpsolve`[4]). When this is the case, the type constraints in $\widehat{\varphi}$ are satisfiable if, for each type constraint of the form $\mathsf{T} = \mathsf{S}$, either T or S are type variables, or T and S have the same topmost constructor, *i.e.* they are either both `int`, or both unlimited/linear channel types with the same polarity, or both product types.

The *synthesis phase* computes an assignment that satisfies φ. This is found by applying σ_{int} to all the type constraints in $\widehat{\varphi}$, by choosing a canonical constraint of the form $\alpha = \mathsf{T}$ where T is proper for each $\alpha \in \mathsf{dom}(\widehat{\varphi})$, and then by solving the resulting system $\{\alpha_i = \mathsf{T}_i\}$ of equations. By [4, Theorem 4.2.1], this system has exactly one solution σ_{type} and now $\sigma_{int} \cup \sigma_{type} \vDash \varphi$. There may be type variables α for which there is no $\alpha = \mathsf{T}$ constraint with T proper. These type variables denote values not used by the process, like a message that is received from one channel and just forwarded on another one. These variables are assigned a type that can be computed canonically.

Example 5.1. The constraints shown in Table 3 entail $0 \leq \theta_3 - k$ and $0 \leq \theta_5 - k$ and $0 \leq \theta_6 - k$ namely $k \leq \theta_3$ and $k \leq \theta_5$ and $k \leq \theta_6$ must hold. When $k = 0$, these constraints can be trivially satisfied by assigning 0 to all ticket variables. When $k = 1$, from the type of a at step (4) of the reconstruction algorithm we deduce that a must have at least 2 tickets. Indeed, a is sent in two messages. It is only considering the remaining processes $c!(e,f)$ and $c!(f,e)$ that we learn that y is instantiated with a. Then, a needs one more ticket, to account for the further and last travel in the recursive invocation $c!(a,y)$. ■

6 Concluding remarks

A key distinguishing feature of the type systems in [18] is the use of polymorphic recursion. Type reconstruction in presence of polymorphic recursion is notoriously undecidable [10,7]. In our case, polymorphism solely concerns levels and reconstruction turns out to be doable. A similar situation is known for *effect systems* [1], where polymorphic recursion restricted to effects does not prevent complete type reconstruction [2].

We have conducted some benchmarks on generalizations of Example 2.1 to N-dimensional hypercubes of processes using full-duplex communication. The table below reports the reconstruction times for the analysis of an hypercube of side 5 and N varying from 1 to 4. The table details the dimension, the number of processes and channels, and the times (in seconds) spent for linearity analysis [19], constraint generation (Section 4) and saturation, solution of level and ticket constraints (Section 5). The

[3] `http://www.gnu.org/software/glpk/`
[4] `http://sourceforge.net/projects/lpsolve/`

solver used for level and ticket constraints is `GLPK` 4.48 and times were measured on a 13" MacBook Air running a 1.8 GHz Intel Core i5 with 4 GB of 1600MHz DDR3.

N	**Processes**	**Channels**	**Linearity**	**Gen.+Sat.**	**Levels**	**Tickets**	**Overall**
1	5	8	0.021	0.006	0.002	0.003	0.032
2	25	80	0.128	0.051	0.009	0.012	0.200
3	125	600	1.439	0.844	0.069	0.124	2.477
4	625	4000	33.803	26.422	1.116	3.913	65.254

Reconstruction times scale almost linearly in the number of channels as long as there is enough free main memory. With $N = 4$, however, the used memory exceeds 10GB causing severe memory (de)compression and swapping. The running time inflates consequently. We have not determined yet the precise causes of such disproportionate consumption of memory, which the algorithms do not seem to imply. We suspect that they are linked to our naive implementation of the algorithms in a lazy language (Haskell), but a more rigorous profiling analysis is left for future investigation. Integer programming problems are NP-hard in general, but the time used for integer constraint resolution appears negligible compared to the other phases. As suggested by one reviewer, the particular nature of such constraints indicates that there might be more clever way of solving them, for example by using SMT solvers.

Our work has been inspired by previous type systems ensuring (dead)lock freedom for generic π-calculus processes [11,13] and corresponding type reconstruction algorithms [12]. These type systems and ours are incomparable: [11,13] use sophisticated behavioral types that provide better accuracy with respect to unlimited channels as used for modeling mutual exclusion and concurrent objects. On the other hand, our type systems exploit level polymorphism for dealing with recursive processes in cyclic topologies, often arising in the modeling of parallel algorithms and sessions. Whether and how the strengths of both approaches can be combined together is left for future research. A more thorough comparison between these works can be found in [18].

There is a substantial methodological difference between our approach and those addressing sessions, particularly *multiparty sessions* [8,6]. Session-based approaches are *top down* and *type driven*: types/protocols come first, and are used as a guidance for developing programs that follow them. These approaches guarantee *by design* a number of properties, among which (dead)lock freedom when different sessions are not interleaved. Our approach is *bottom up* and *program driven*: programs come first, and are used for inferring types/protocols. The two approaches can integrate and complement each other. For example, type reconstruction may assist in the verification of legacy or third-party code (for which no type information is available) or for checking the impact of code changes due to refactoring and/or debugging. Also, some protocols are hard to describe *a priori*. For example, describing the essence of full-duplex communications (Example 2.1) is far from trivial [6]. In general, processes making use of channel mobility (delegation) and session interleaving, or dynamic network topologies with variable number of processes, are supported by our approach (within the limits imposed by the type systems), but are challenging to handle in top-down approaches. Inference of progress properties akin to lock freedom for session-based calculi has been studied in [17,3], although only finite types are considered in these works.

The reconstruction of *global* protocol descriptions from *local* session types has been studied in [15,16]. In this respect, our work fills the remaining gap and provides a reconstruction tool from processes to local session types. We plan to investigate the integration with [15,16] in future work.

Acknowledgments. The authors are grateful to the reviewers for their detailed comments and useful suggestions. The first two authors have been supported by Ateneo/CSP project SALT. The first author has also been supported by ICT COST Action IC1201 BETTY and MIUR project CINA.

References

1. T. Amtoft, F. Nielson, and H. Nielson. *Type and effect systems: behaviours for concurrency*. Imperial College Press, 1999.
2. T. Amtoft, F. Nielson, and H. R. Nielson. Type and behaviour reconstruction for higher-order concurrent programs. *J. Funct. Program.*, 7(3):321–347, 1997.
3. M. Coppo, M. Dezani-Ciancaglini, L. Padovani, and N. Yoshida. Inference of global progress properties for dynamically interleaved multiparty sessions. In *COORDINATION'13*, LNCS 7890, pages 45–59. Springer, 2013.
4. B. Courcelle. Fundamental properties of infinite trees. *Theor. Comp. Sci.*, 25:95–169, 1983.
5. O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In *PPDP'12*, pages 139–150. ACM, 2012.
6. P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP'12*, LNCS 7211, pages 194–213. Springer, 2012.
7. F. Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, Apr. 1993.
8. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.
9. A. Igarashi and N. Kobayashi. Type reconstruction for linear π-calculus with I/O subtyping. *Inf. and Comp.*, 161(1):1–44, 2000.
10. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):290–311, 1993.
11. N. Kobayashi. A type system for lock-free processes. *Inf. and Comp.*, 177(2):122–159, 2002.
12. N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.
13. N. Kobayashi. A new type system for deadlock-free processes. In *CONCUR'06*, LNCS 4137, pages 233–247. Springer, 2006.
14. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999.
15. J. Lange and E. Tuosto. Synthesising choreographies from local session types. In *CONCUR'12*, LNCS 7454, pages 225–239. Springer, 2012.
16. J. Lange, E. Tuosto, and N. Yoshida. From communicating machines to graphical choreographies. In *POPL'15*, pages 221–232. ACM, 2015.
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. L. Padovani. Deadlock and lock freedom in the linear π-calculus. In *CSL-LICS'14*, pages 72:1–72:10. ACM, 2014. `http://hal.archives-ouvertes.fr/hal-00932356v2/`.
19. 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.