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

Luca Padovani

Dipartimento di Informatica, Università di Torino, Italy

Abstract. We extend the linear π-calculus with composite and equi-recursive types in a way that enables the sharing of data containing linear values, provided that there is no overlapping access on such values. We show that the extended type system admits a complete type reconstruction algorithm and, as a by-product, we solve the problem of reconstruction for equi-recursive session types.

1 Introduction

The linear π-calculus [11] is a formal model of communicating processes in which channels are either *unlimited* or *linear*. Unlimited channels can be used without restrictions, while linear channels can only be used once for an input/output. Linear channels occur frequently in actual systems, they allow optimisations and efficient implementations [6,5,11], and communications on linear channels enjoy important properties such as interference freedom and partial confluence [13,11].

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. It is a known fact, however, that more sophisticated interactions can be implemented taking advantage of *channel mobility* using a continuation-passing style [9,2]. The basic idea is that, along with the proper payload of a communication, one can send another channel on which the rest of the conversation takes place. This technique is illustrated below

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

where a *producer* process P sends messages to a *consumer* process 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 modelled (1.1) re-using the same channel x 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.

A. Muscholl (Ed.): FOSSACS 2014, LNCS 8412, pp. 88–102, 2014.

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 be a channel type satisfying the equation $t = [\mathtt{int} \times t]^{1,0}$ where $\mathtt{int} \times t$ is the type of messages received from the channel and 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 is never used for input and is used once for output. Finally, there are two (non-binding) occurrences of a in the producer: a in $\overline{x}\langle n, a\rangle$ 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 at the next iteration. 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.

Note that linear channels, like a in (1.1), may syntactically occur multiple times and that 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 can be seen as the *combination* of t and s as defined in [11,14].

One legitimate question is whether and how the notions of type compatibility and combination extend beyond channel types. In this respect, the existing literature lacks definitive and satisfactory answers: the works on (type reconstruction for) the linear π-calculus [11,7] do not consider composite or recursive types. Linear type systems with composite types have been discussed in [5,6] for the linear π-calculus and in [15] for a functional language. These works, however, see linearity as a "contagious" property: every structure that contains linear values becomes linear itself (there are a few exceptions for specific types [10] or relaxed notions of linearity [8]). 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 share the *same* composite data structure, provided that they access *different* linear values stored therein. The problem is whether the type system is accurate enough to capture the fact that there are no overlapping accesses to the same linear values. To illustrate, consider the type t_{list} 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 typing lists of linear channels with type $[\mathtt{int}]^{0,1}$. If we follow [15,5,6], an identifier l having type t_{list} can syntactically occur only once in a program, and a process like for instance $Odd\langle l\rangle \mid Even\langle l\rangle$ where l occurs twice is illegal. However, suppose that *Odd* and *Even* are the two processes defined by

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

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*:

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

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 combination was $[\mathtt{int} \times t]^{1,1}$, 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 combination is t_{list}. The "only" difference is that, while t and s were channel types and their combination could be expressed simply by combining the uses in them, t_{odd} and t_{even} are recursive, structured types that combine to t_{list} *in the limit*.

To summarise, composite and recursive types are basic yet fundamental features whose smooth integration in the linear π-calculus requires some care. In this work we extend the linear π-calculus with composite and recursive types in such a way that multiple processes can safely share structured data containing linear values and we define a complete type reconstruction algorithm for the extended type system.

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 on the author's home page.

2 The Linear π-Calculus

We let $m, n, \ldots$ range over integer numbers; we use a countable set of *channels* $a, b, \ldots$ and a disjoint countable set of *variables* $x, y, \ldots$; *names* $u, v, \ldots$ 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:

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

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 $i \in \{\mathtt{inl}, \mathtt{inr}\}$. *Values* $\mathsf{v}, \mathsf{w}, \ldots$ are expressions without variables.

Processes $P, Q, \ldots$ comprise the standard constructs of the asynchronous π-calculus. 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.[1] The process $\mathtt{case}\ e\ \mathtt{of}\{i\ x_i \Rightarrow$

[1] As pointed out by one reviewer, this construct is superfluous, because the type system we are about to define allows linear pairs to be accessed more than once with the conventional projection operators. We have added support for pair projections in the implementation, but we keep the `let` construct in the formal presentation.

Table 1. Reduction of processes

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

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

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

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

$P_i\}_{i=\mathtt{inl},\mathtt{inr}}$ 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 v for the free occurrences of x in P.

The operational semantics of the language is defined in terms of a structural congruence relation 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, like in [11], we decorate the relation with *labels* ℓ that are either channels or the special symbol τ, denoting an unobservable action: in [R-COMM] the label is 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 escapes the scope of 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 $\mathtt{int}$, products $t_1 \times t_2$ typing values of the form $(\mathsf{v}_1,\mathsf{v}_2)$ where v_i has type t_i for $i = 1,2$, and disjoint sums $t_1 \oplus t_2$ typing values of the form $(\mathtt{inl}\ \mathsf{v})$ where v has type t_1 or of the form $(\mathtt{inr}\ \mathsf{v})$ where 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 none, 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 μ operators are used for building recursive types, as usual. Notions 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 let $\mu\alpha.t = t\{\mu\alpha.t/\alpha\}$ 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

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

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. Note that $1 \not\leq \omega$ does *not* hold. 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 coinductively 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 $[t]^{\kappa_1,\kappa_2}\ \mathscr{R}_{type}\ [s]^{\kappa_3,\kappa_4}$ implies $t = s$ regardless of $\mathscr{R}_{type}$. The relation $t \asymp t$ is particularly interesting, because it characterises unlimited types, those typing 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 example, $[t]^{0,0} \times [t]^{1,0}$ is linear while $[t]^{0,0} \times [t]^{\omega,0}$ is unlimited. Recursion does not affect the classification of types into linear and unlimited. For example, $\mu\alpha.[\texttt{int} \times \alpha]^{1,0}$ 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:

$$\begin{aligned} \Gamma_1 + \Gamma_2 &\stackrel{\text{def}}{=} \Gamma_1, \Gamma_2 && \text{if } \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \emptyset \\ (\Gamma_1, u : t) + (\Gamma_2, u : s) &\stackrel{\text{def}}{=} (\Gamma_1 + \Gamma_2), u : t \vee s && \text{if } t \asymp s \end{aligned} \tag{2.3}$$

The operator $+$ generalises type combination in [11] and the $\uplus$ operator in [14]. 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

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

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

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

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

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

$\mathsf{dom}(\Gamma_1 + \Gamma_2) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$. 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 is used by a process for sending a message of type `int`, it has type $[\mathtt{int}]^{0,1}$ in that process; if it is used by another process for receiving a message of type `int`, it has type $[\mathtt{int}]^{1,0}$ in that process. Overall, the two processes in parallel use the channel 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. These rules are basically the same as those found in the literature [11,7], and the additional flexibility enabled by our typing discipline is actually a consequence of our 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 ($\Gamma \asymp \Gamma$). 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 Γ provided that P is well typed in Γ and that Γ is unlimited. The rationale for this is that $*P$ stands for an unbounded number of copies of P composed in parallel, hence it cannot contain (free) linear channels. The rules [T-PAR], [T-LET], and [T-CASE] are conventional. 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 sensible choice in practice.

The type system is sound and the results in Section 4.3 of [11] 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

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

and we assume that $\texttt{inl}\ 0$ represents the empty list $\texttt{[]}$ and $\texttt{inr}\ (y,z)$ 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):

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

$$\dfrac{a : [t_{odd}]^{\omega,0}, b : [t_{even}]^{0,\omega} \vdash a(x).\texttt{case}\ x\ \texttt{of}\ \cdots}{a : [t_{odd}]^{\omega,0}, b : [t_{even}]^{0,\omega} \vdash *a(x).\texttt{case}\ x\ \texttt{of}\ \cdots}\,[\text{T-REP}]}\,[\text{T-IN}]$$

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 derivation 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

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

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

3 Constraint Generation

We can formalise the problem of type reconstruction as follows: given a process P, find a type environment Γ such that $\Gamma \vdash P$, provided there is one. In general we also want to maximise the number of linear types in Γ. The rules shown in Table 2 rely on a fair amount of guessing that concerns 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 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 generalise 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, respectively, and we say that t is *proper* if it is not 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 such that $\kappa_1 \sim \kappa_2$ holds for all κ_1 and κ_2. We call $\kappa_1 \mathrel{\mathscr{R}_{\mathsf{c}}} \kappa_2$ a *use constraint* and $t_1 \mathrel{\mathscr{R}_{\mathsf{c}}} t_2$ a *type constraint*. 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 ensuring type and use combination, as well as for expressing the assumption that a type/use is unlimited (see *e.g.* the premise of [T-IDLE]). Finally, $\sim_{\mathsf{c}}$ constraints relate types that must be *structurally coherent*. For example, $[t]^{\kappa_1,\kappa_2} \sim [t]^{\kappa_3,\kappa_4}$ holds regardless of κ_1, κ_2, κ_3, and κ_4 (but note that according to (2.2) there must be the same t in the two types). We will see in Section 4 the role of these 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, respectively. 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 $\kappa_1 \mathrel{\mathscr{R}_{\mathsf{c}}} \kappa_2 \in \mathscr{C}$. We extend the $\leq$ relation pointwise to solutions and we say that a solution σ for $\mathscr{C}$ is *minimal* if every solution $\sigma' \leq \sigma$ for $\mathscr{C}$ is such that $\sigma \leq \sigma'$. 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 type environments Γ_1 and Γ_2 and produce a pair consisting of another type environment Γ and a set of constraints $\mathscr{C}$:

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

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

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 (2.3). 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

Table 3. Constraint generation for expressions and processes

Expressions

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

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

Processes

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

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

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

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

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 reconstruct type environments and to generate constraints are presented in Table 3 and derive judgements 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 $+$ 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] 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 $0 <_{\mathsf{c}} \rho$ 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 the 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 [I-WEAK] may be necessary are easy to spot (in the premises of all the aforementioned rules for the binding constructs). What we gain with [I-WEAK] is a simpler presentation of the rules for constraint generation.

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 minimal 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

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

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

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

where

$$\begin{array}{lll}
\mathscr{C}_1 \stackrel{\text{def}}{=} \{0 <_c \rho_1\} & \mathscr{C}_2 \stackrel{\text{def}}{=} \mathscr{C}_1 \cup \{\gamma \asymp_c \gamma\} & \mathscr{C}_3 \stackrel{\text{def}}{=} \mathscr{C}_2 \cup \{\alpha_1 =_c [\texttt{int}]^{0,\rho_1} \times \gamma\} \\
\mathscr{C}_4 \stackrel{\text{def}}{=} \{0 <_c \rho_2\} & \mathscr{C}_5 \stackrel{\text{def}}{=} \mathscr{C}_4 \cup \{\beta \asymp_c \beta\} & \mathscr{C}_6 \stackrel{\text{def}}{=} \mathscr{C}_5 \cup \{\alpha_2 =_c \beta \times [\texttt{int}]^{0,\rho_2}\} \\
\mathscr{C}_7 \stackrel{\text{def}}{=} \mathscr{C}_3 \cup \mathscr{C}_6 \cup \{\alpha_1 \asymp_c \alpha_2, \alpha_1 \leq_c \alpha, \alpha_2 \leq_c \alpha\} & \mathscr{C}_8 \stackrel{\text{def}}{=} \mathscr{C}_7 \cup \{0 <_c \rho_3\}
\end{array}$$

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. ■

Table 4. Constraint solver algorithm

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

1. Compute $\overline{\mathscr{C}}$;
2. Compute a minimal solution σ_{use} 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 have different topmost type constructors, then **fail**;
4. Let $\sigma_{type} = \{\alpha \mapsto \mathsf{sup}_{\overline{\mathscr{C}},\sigma_{use}}(\{\alpha\}) \mid \alpha \in \mathsf{dom}(\overline{\mathscr{C}})\}$;
5. Return $\sigma_{use} \cup \sigma_{type}$.

4 Constraint Solving

In this section we define an algorithm that determines whether a given set of constraints $\mathscr{C}$ is satisfiable and, if this is the case, 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, we *saturate* the set $\mathscr{C}$ with all the constraints that are entailed by those already in $\mathscr{C}$. Entailment is expressed through a binary relation $\vDash$ defined as follows:

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

The first three rules [E-REFL], [E-SYMM], and [E-TRANS] compute the reflexive, symmetric, and transitive closures of those relations that enjoy such properties. Rule [E-COMP 1] propagates compatibility constraints across 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 types 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 $\sim_{\mathsf{c}}$ constraints between any pair of related type expressions. This is necessary to make sure that all message type equality constraints 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 entailment 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 whether $\overline{\mathscr{C}}$ 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 satisfiability of use and type constraints and fail if any of these checks fails.

Recall that there is a finite number of use constraints, which are relationships 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 define an "optimal" algorithm that aims at maximising the number of use variables that are assigned value 1 as opposed to ω. 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, $\mathtt{int} \leq_{\mathsf{c}} [\alpha]^{\kappa_1,\kappa_2}$ 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 unsatisfiable 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_{type} for the type constraints in $\overline{\mathscr{C}}$ given any minimal solution σ_{use} for the use constraints in $\overline{\mathscr{C}}$ determined at step 2 of the algorithm. To compute σ_{type} we need the definitions 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 share the same topmost type constructor with some $t \in T$ but have possibly different uses. 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 $\mathsf{sup}_{\overline{\mathscr{C}},\sigma_{use}}(\{\alpha\})$ 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 $\sim_{\mathsf{c}}$ 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 $\sim_{\mathsf{c}}$ 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 $\mathsf{zero}_{\mathscr{C},\sigma}$).

Interpreting $\mathsf{sup}_{\mathscr{C},\sigma}$ and $\mathsf{zero}_{\mathscr{C},\sigma}$ as functions is appropriate for presentation (and implementation) purposes, but formally tricky 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 formal 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:

- 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 $\mathsf{sup}_{\overline{\mathscr{C}},\sigma_{use}}$ in Table 4 regard well-defined equations. We conclude:

Theorem 4.1. *If the algorithm returns σ, then σ is a minimal 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 $\sigma_{use} \stackrel{\text{def}}{=} \{\rho_1 \mapsto 1, \rho_2 \mapsto 1, \rho_3 \mapsto 1\}$. From this we obtain

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

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 channels contained therein is used exactly once. ■

5 Concluding Remarks

Previous works on the linear π-calculus either ignore composite types [11,7] 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 complete 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 different type composition operator. This small change has nonetheless nontrivial 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 these problems by expressing type combination in previous works (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 $t_i \leq t_3$, and by seeking for *minimal* solutions of the constraint set. Our extension also gives renewed relevance to types like $[t]^{0,0}$. In previous works these types were admitted but essentially useless: channels with such types could only be passed around in messages without actually ever being used. That is, they could be erased without affecting processes. In our type system, it is the existence of these types that enables the sharing of structured data (see the decomposition of t_{list} into t_{even} and t_{odd} in Section 1).

Given that sessions [3,4] can be fully encoded into the linear π-calculus [9,2], we also indirectly provide a complete reconstruction algorithm for equi-recursive session types without subtyping. Interestingly, the concept of *duality*, which is a source of significant complication of the type reconstruction algorithm for *finite* session types [12], is simplified by the encoding, where it reduces to compatibility.

To assess the feasibility of the approach, we have developed a prototype based on the naïve constraint saturation and combinatorial verification of use expressions described in Section 4. Regarding the complexity of the reconstruction algorithm, the most critical

aspects are the solution of use constraints and the computation of the transitive closure of type constraints. While polynomial algorithms are known for the latter, the former problem entails, in principle, an exponential cost. Preliminary experiments with the prototype implementation of the algorithm have shown that, in both cases, constraints can often be partitioned into relatively small independent subsets that can be solved in isolation (the prototype already supports such partitioning for use constraints). This property paves the way for significant performance improvements.

Acknowledgements. The author is grateful to the FoSSaCS'14 reviewers for their detailed and insightful comments. This work has been partially supported by ICT COST Action IC1201 BETTY, MIUR project CINA, and Ateneo/CSP project SALT.

References

1. Courcelle, B.: Fundamental properties of infinite trees. Theor. Comp. Sci. 25, 95–169 (1983)
2. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. In: PPDP 2012, pp. 139–150. ACM (2012)
3. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993)
4. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
5. Igarashi, A.: Type-based analysis of usage of values for concurrent programming languages (1997), `http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/papers/`
6. Igarashi, A., Kobayashi, N.: Type-based analysis of communication for concurrent programming languages. In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302, pp. 187–201. Springer, Heidelberg (1997)
7. Igarashi, A., Kobayashi, N.: Type Reconstruction for Linear π-Calculus with I/O Subtyping. Inf. and Comp. 161(1), 1–44 (2000)
8. Kobayashi, N.: Quasi-linear types. In: POPL 1999, pp. 29–42. ACM (1999)
9. Kobayashi, N.: Type systems for concurrent programs. In: Aichernig, B.K. (ed.) Formal Methods at the Crossroads. From Panacea to Foundational Support. LNCS, vol. 2757, pp. 439–453. Springer, Heidelberg (2003), Extended version at `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf`
10. Kobayashi, N.: A new type system for deadlock-free processes. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 233–247. Springer, Heidelberg (2006)
11. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the pi-calculus. ACM Trans. Program. Lang. Syst. 21(5), 914–947 (1999)
12. Mezzina, L.G.: How to infer finite session types in a calculus of services and sessions. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 216–231. Springer, Heidelberg (2008)
13. Nestmann, U., Steffen, M.: Typing confluence. In: FMICS 1997, pp. 77–101 (1997), Also available as report ERCIM-10/97-R052, European Research Consortium for Informatics and Mathematics (1997)
14. Sangiorgi, D., Walker, D.: The Pi-Calculus - A theory of mobile processes. Cambridge University Press (2001)
15. Turner, D.N., Wadler, P., Mossin, C.: Once upon a type. In: FPCA 1995, pp. 1–11 (1995)