Aalborg Universitet

Using session types for reasoning about boundedness in the π-calculus

Hüttel, Hans

Published in:
Electronic Proceedings in Theoretical Computer Science, EPTCS

DOI (link to publication from Publisher):
10.4204/EPTCS.255.5

Creative Commons License
CC BY 4.0

Publication date:
2017

Document Version
Publisher's PDF, also known as Version of record

Link to publication from Aalborg University

Citation for published version (APA):
Hüttel, H. (2017). Using session types for reasoning about boundedness in the π-calculus. *Electronic Proceedings in Theoretical Computer Science, EPTCS*, *255*, 67-82. https://doi.org/10.4204/EPTCS.255.5

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal -

Take down policy
If you believe that this document breaches copyright please contact us at vbn@aub.aau.dk providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from vbn.aau.dk on: December 04, 2025

Using Session Types for Reasoning About Boundedness in the π-Calculus

Hans Hüttel*
Department of Computer Science, Aalborg University, Denmark

The classes of depth-bounded and name-bounded processes are fragments of the π-calculus for which some of the decision problems that are undecidable for the full calculus become decidable. P is depth-bounded at level k if every reduction sequence for P contains successor processes with at most k active nested restrictions. P is name-bounded at level k if every reduction sequence for P contains successor processes with at most k active bound names. Membership of these classes of processes is undecidable. In this paper we use binary session types to decise two type systems that give a sound characterization of the properties: If a process is well-typed in our first system, it is depth-bounded. If a process is well-typed in our second, more restrictive type system, it will also be name-bounded.

1 Introduction

In the π-calculus, the notion of name restriction is particularly important. The study of properties of name binding is a testbed for studying properties of bindable entities and notions of scoping in programming languages. In a restriction process $(\nu x)P$ the name x has P as its scope and it is customary to think of x as a new name, known only to P. It is the interplay between restriction and replication (or recursion) that leads to the π-calculus being Turing-powerful. Without either of these two constructs, this is no longer the case [9].

With full Turing power comes undecidability of commonly encountered decision problems such as the termination problem "Given process P, will P terminate?" and the coverability problem "Given process P and process Q, is there a computation of P that will eventually reach a process that has Q as a subprocess?". Several classes of processes have been identified for which (some of) these problems remain decidable. Examples are the *finitary processes* without replication or recursion, the *finite-control processes* [3] in which every process has a uniform bound on the number of parallel components in any computation, the *bounded processes* [2] for which there are only finitely many successors of any reduction up to a special notion of structural congruence with permutation over a finite set of names, and *processes with unique receiver and bounded input* [1].

More recently, there has been work in this area that studies limitations on the use of restriction that will ensure decidability. The notion of *depth-bounded* processes was introduced by Meyer in [11]. A process P is depth-bounded at level k if there is an upper bound k, such that any reduction sequence for P will only lead to successor processes that have at most k active nested restrictions – that is, restrictions not occurring underneath some prefix. Termination and coverability are both decidable for depth-bounded processes. The class of depth-bounded processes is expressive and contains a variety of other decidable subsets of the π-calculus. Moreover, for any fixed k it is decidable if a process P is depth-bounded at level k; however, it is undecidable if there exists a k for which P is depth-bounded [11].

*E-mail: hans@cs.aau.dk

K. Peters and S. Tini (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2017).
EPTCS 255, 2017, pp. 67–82, doi:10.4204/EPTCS.255.5

© Hans Hüttel
This work is licensed under the Creative Commons Attribution License.

In a more recent paper [4], D'Osualdo and Ong have introduced a type system that gives a sound characterization of depth-boundedness: If P is well-typed, then P is depth-bounded. The underlying idea of this type system is to analyze properties of the hierarchy of restrictions within a process.

Another class of π-calculus processes is that of *name-bounded* processes, introduced by Hüchting et al. [8]. A process P is name-bounded at level k if any reduction sequence for P will only lead to successor processes with at most k active bound names.

The goal of this paper is to use binary session types [7] to give sound characterizations of depth-boundedness, respectively name-boundedness in the π-calculus: If a process is well-typed, we know that it is depth-bounded, respectively name-bounded. The advantages of this approach are the following: Firstly, unlike the type system proposed by D'Osualdo and Ong [4] we can directly keep track of how names are used and where they appear in a process, since this is central to session type disciplines. The linear nature of session names ensures that every name of this kind will always, when used, occur in precisely two parallel components. Secondly, the session type disciplines are resource-conscious; we can therefore ensure that new bound names are only introduced whenever existing bound names can no longer be used. Both type systems use finite session types to achieve this for recursive processes. Informally, a new recursive call can only occur once all sessions involving the bound names of the current recursive call have been used up. In the proof of the soundness of the system for characterizing name-boundedness system, we make use of the fact that it is a more restrictive version of that for depth-boundedness.

The rest of our paper is organized as follows. Section 2 describes the π-calculus that we will consider; section 3 introduces the notions of boundedness. Section 4 presents a type system for depth-bounded processes, which is analyzed in sections 5 and 6. Section 7 presents a type system for name-bounded processes. Section 8 discusses the relationship with other classes of processes.

2 A typed π-calculus with recursion

We follow Meyer [11] and use a π-calculus with *recursion* instead of replication. The reason behind this choice of syntax is that we would like infinite behaviours to make use of bound names in a non-trivial manner that guarantees boundedness properties. In general, the combination of restriction and replication in $!(\nu x)P$ will result in a process that fails to be name-bounded.

2.1 Syntax

We assume the existence of a countably infinite set of names, $\mathscr{N}$, and let $a,b,\ldots$ and $x,y,\ldots$ range over $\mathscr{N}$. Moreover, we assume a countably infinite set of recursion variables, $\mathscr{R}$, and let $X,Y,\ldots$ range over $\mathscr{R}$.

2.1.1 Processes

Following [5] we will use a version of the π-calculus with *polarized names* in order to ensure that the endpoints of a channel will not end up in the same parallel component. We assume polarities ranged over by $p,q\ldots$. The polarities $+$ and $-$ are dual; we define $\overline{+}=-$ and $\overline{-}=+$. The empty polarity ε is self-dual and used for names used as channels that are not session channels and to tag name occurrences in the binding constructs of input and restriction. We call the set of polarized names $\mathscr{N}_{\text{pol}}$.

The formation rules of processes are given by

$$P ::= x^p(y).P_1 \mid \overline{x^p}\langle y^q\rangle.P_1 \mid P_1 \mid P_2 \mid (\nu x : T)P_1 \mid \mu X.P_1 \mid X \mid \mathbf{0}$$
$$p ::= + \mid - \mid \varepsilon$$

As usual, $x^p(y).P_1$ denotes a process that inputs a name on channel x and continues as P_1; the unpolarized name y is bound in P_1. $\overline{x^p}\langle y^q\rangle.P_1$ is a process that outputs the polarized name y^q on channel x and continues as P_1. $P_1 \mid P_2$ is the parallel execution of P_1 and P_2. $\mu X.P_1$ is a recursive process with body P_1. We assume that every such recursive process is *guarded*; every occurrence of a recursion variable must be found underneath an input or an output prefix. In $\mu X.P_1$ the μX is called a binding occurrence of X. A process P is *recursion-closed* if every recursion variable X in P has a binding occurrence for some subprocess $\mu X.P_1$ and if all recursion variables are distinct. We employ a notion of typed restriction, which we will now explain.

2.1.2 Typed restrictions

In the restriction $(\nu x : T)P_1$ the unpolarized name x is bound in P_1 and annotated with type T. Our set of types $\mathscr{T}$ is a non-recursive version of the binary session types introduced by Gay and Hole [5] and defined by the formation rules

$$T ::= S \mid \mathsf{Ch}(T)$$
$$S ::= (S_1, S_2) \mid !T.S \mid ?T.S \mid \mathsf{end}$$

A type T can be a *linear* endpoint type S or pair of endpoints (S_1, S_2), or an *unlimited* channel type $\mathsf{Ch}(T)$. An endpoint type S of the form $!T.S$ denotes that a channel of this type can output a name of type T; afterwards, the channel will have type S. An endpoint type of the form $?T.S$ denotes that a channel of this type can input a name of type T; afterwards, the channel will have type S. The special endpoint type end is the type of an endpoint that allows no further communication. If $T = (!T_1.S_2, ?T_1'.S_2')$ we let $T \downarrow= (S_2, S_2')$; this denotes the successor of a pair of endpoint types. If $T = \mathsf{Ch}(T_1)$, then $T \downarrow= T$.

We use the type annotation of restrictions to keep track of the subject name that led to a reduction and of how the types of bound names evolve.

The sets of free and bound names of a process, $\mathrm{fn}(P)$ and $\mathrm{bn}(P)$, are defined as usual. To simplify the presentation, we assume all free and bound names distinct. We let $P\{^y/_x\}$ denote the capture-avoiding substitution that replaces all free occurrences of x in P by y. A name $n \in \mathrm{bn}(P)$ is *active* if it does not appear underneath a prefix.

2.1.3 Structural congruence

Structural congruence is the least congruence relation for the process constructs that is closed under the axioms in Table 1.

Following Meyer [11], we sometimes consider processes in *restricted form*. A process is in inner normal form, if every restriction $(\nu x : T)$ only encloses parallel components that contain x. A process is in outer normal form if every restriction not underneath a prefix appears at the outermost level.

Definition 1 (Normal forms)**.** Let P be a process.

- P is in *inner normal form* if for every subprocess $(\nu x : T)(P_1 \mid \cdots \mid P_k)$ where none of the P_i are parallel compositions of processes, we have $x \in \mathrm{fn}(P_i)$ for all $1 \leq i \leq k$.

(NEW-1)	$(\nu x:T)(\nu y:T')P \equiv (\nu y:T')(\nu x:T)P$	(NIL-1)	$P \mid \mathbf{0} \equiv P$
(NEW-2)	$(\nu x:T)P \mid Q \equiv (\nu x:T)(P \mid Q)$ if $x \notin \mathrm{fn}(Q)$	(NIL-2)	$(\nu x:T)\mathbf{0} \equiv \mathbf{0}$
(PAR-1)	$P \mid Q \equiv Q \mid P$		
(PAR-2)	$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$		

Table 1: Structural congruence: Axioms and rules

(COM-ANNOT) $$a^p(x).P_1 \mid \overline{a^{\overline{p}}}\langle y^q\rangle.P_2 \xrightarrow{\{a\}} P_1\{^{y^q}/_x\} \mid P_2$$

(PAR-ANNOT) $$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

(NEW-ANNOT) $$\frac{P \xrightarrow{\alpha} P'}{(\nu x:T)P \xrightarrow{\alpha} (\nu x:T')P'} \quad \text{where} \quad \begin{array}{l} T = T' \text{ if } x \notin \alpha \\ T' = T\downarrow \text{ if } x \in \alpha \end{array}$$

(UNFOLD-ANNOT) $$\frac{P > Q \quad Q \xrightarrow{\alpha} P'}{P \xrightarrow{\{\mathsf{rec}\}\cup\alpha} P'}$$

(STRUCT-ANNOT) $$\frac{P \equiv Q \quad Q \xrightarrow{\alpha} Q' \quad Q' \equiv P'}{P \xrightarrow{\alpha} P'}$$

Table 2: Annotated reduction rules

- P is in *outer normal form* if $P = (\nu x_1)\ldots(\nu x_k)P_1$ such that $x_i \in \mathrm{fn}(P_1)$ for all $1 \le i \le k$ and such that all restrictions in P_1 appear underneath prefixes.

Proposition 1. *For every process P we can construct a process $P_1 \equiv P$ in inner normal form and a process $P_2 \equiv P$ in outer normal form.*

2.2 An annotated reduction semantics

We define the behaviour of processes by an annotated reduction semantics that keeps track of when recursive unfoldings are necessary. Reductions are of the form $P \xrightarrow{\alpha} P'$ where either $\alpha = \{a\}, a \in \mathcal{N}$ or $\alpha = \{\mathsf{rec}, a\}$ for $a \in \mathcal{N}$. The latter annotation indicates that recursive unfolding was necessary to obtain the reduction. We define $\mathrm{n}(\{a\}) = a$ and $\mathrm{n}(\{\mathsf{rec}, a\}) = a$. The reduction rules are found in Table 2. Note that in the rule (NEW-ANNOT) the type associated with the bound name x evolves, if x if x is responsible for the communication and T is a session type.

If P reduces to P' in zero or more reduction steps, we write $P \to^* P'$.

Recursion is described by an unfolding relation which we define in Table 3. In the definition, we use the notion of *unfolding contexts*. An unfolding context $C[\,]$ is an incomplete process terms whose hole indicates where a prefix that participates in a reduction step appears as the direct result of unfolding a

recursive process.

Definition 2 (Unfolding contexts)**.** The set of unfolding contexts is given by the formation rules

$$C ::= [\,] \mid P \mid (\nu x : T)C$$

$$\text{(UNFOLD)} \quad \mu X.P > P[\mu X.P/X] \qquad\qquad \text{(CONTEXT)} \quad \frac{P > P'}{C[P] > C[P']}$$

Table 3: The rules for unfolding

Example 1. We can write the process

$$P \stackrel{\text{def}}{=} (\nu c : T)\mu X.a(x).\overline{x}\langle x\rangle.X \mid \mu Y.(\nu b : U)\overline{a}\langle b\rangle.x(y).Y$$

as

$$C_1[\mu X.a(x).\overline{x}\langle x\rangle.X] \text{ where } C_1 = [(\nu c : T)[] \mid \mu Y.(\nu b : U)\overline{a}\langle b\rangle.x(y).Y]$$

or

$$C_2[\mu Y.(\nu b : U)\overline{a}\langle b\rangle.x(y).Y] \text{ where } C_2 = (\nu c : T)\mu X.a(x).\overline{x}\langle b\rangle.X \mid [].$$

3 Notions of boundedness

Meyer introduces three notions of boundedness [11] for the π-calculus, and we now introduce them.

Depth-bounded processes A process P is depth-bounded if every configuration reachable from it can be rewritten so as to have no more than k nested restrictions. To define this, we first introduce a function $\mathsf{nest}(P)$ that counts the maximal number of active nested restrictions. A restriction is active if it does not occur underneath a prefix – this is similar to [4].

Definition 3. The nest function is defined by the clauses

$$\begin{array}{rlrl}
\mathsf{nest}(\mathbf{0}) = 0 & & \mathsf{nest}(X) = 0 & \\
\mathsf{nest}((\nu x : T)P) = 1 + \mathsf{nest}(P) & & \mathsf{nest}(P_1 \mid P_2) = \max(\mathsf{nest}(P_1), \mathsf{nest}(P_2)) & \\
\mathsf{nest}(\mu X.P_1) = \mathsf{nest}(P_1) & & \mathsf{nest}(x^p(y).P_1) = \mathsf{nest}(\overline{x^p}\langle y^q\rangle.P_1) = 0 &
\end{array}$$

The restriction depth of a process is then the minimal nesting depth up to structural congruence.

Definition 4. The depth of a process P is given by

$$\mathsf{depth}(P) = \min\{\mathsf{nest}(Q) \mid Q \equiv P\}.$$

We define a normalization ordering $\succ$ on processes that removes bound names not found in a process. It is generated by the axiom

$$(\nu x)P \succ P \quad \text{if } x \notin \text{fn}(P)$$

and closed under structural congruence. A process P is *normalized* if it has no superfluous bound names, that is, if $P \not\succ$; we write $P \rightsquigarrow Q$ if $P \succ^* Q$ and Q is normalized. [1]

Definition 5 (Depth-bounded process)**.** A process P is depth-bounded if there is a $k \in \mathbb{N}$ such that for every P' where $P \rightarrow^* P'$ we have that for some P'' with $P'' \equiv P'$ we have $\mathsf{depth}(P') \leq k$.

[1] Note that $(\nu x : T)P \equiv P \quad \text{if } x \notin \text{fn}(P)$ is a derived identity if we include the axiom $(\nu x)\mathbf{0} \equiv \mathbf{0}$.

Name-boundedness A process P is *name-bounded* if there exists a constant $k \in \mathbb{N}$ such that whenever $P \to^* P'$ and $P' \rightsquigarrow P''$, then P'' has at most k restrictions. It is obvious that every name-bounded process is also depth-bounded.

Example 2. The term

$$P_1 = \mu X.(\nu r_1)(\overline{r_1^+}\langle a\rangle.X \mid r_1^-(x).X)$$

is depth-bounded with $\mathsf{depth}(P_1) = 1$. The term

$$P_2 = \mu X.(\nu r_1)(\nu r_2)(\overline{r_1^+}\langle r_2\rangle.X \mid r_1^-(x).X \mid \overline{r_2^+}\langle r_1\rangle \mid r_2^-(x))$$

is depth-bounded with $\mathsf{depth}(P_2) = 2$. Neither P_1 nor P_2 is name-bounded.

Width-boundedness A third notion of boundedness is that of *width-boundedness*. A process P is width-bounded if there exists a constant $k \in \mathbb{N}$ such that whenever $P \to^* P'$ we have that every bound name in P' occurs in at most k parallel components. This coincides with the notion of *fencing* recently used by Lange et al. [10] introduced in their analysis of Go programs.

4 Using session types for depth-boundedness

We now present a session type system that gives a sound characterization of depth-boundedness. Our account of binary session types similar to that used by Gay and Hole [5].

4.1 Types and type environments

Our type judgements are of the form $\Gamma, \Delta \vdash P$, where Γ contains the type bindings of the free polarized names in P. A type judgment is to be read as stating that P is well-behaved using the type information found in the type environment Γ and the recursion environment Δ (explained in Section 4.2).

Definition 6. A type environment Γ is a partial function $\Gamma : \mathscr{N}_{\text{pol}} \rightharpoonup \mathscr{T}$ with finite support.

- Γ is *unlimited* if for every $x \in \mathrm{dom}(\Gamma)$ we have $\Gamma(x) = \mathsf{Ch}(T)$ for some T or $\Gamma(x) = \mathsf{end}$
- Γ is *linear* if for every $x \in \mathrm{dom}(\Gamma)$ we have that $\Gamma(x) \neq \mathsf{Ch}(T)$ for all T. We let Γ_{lin} denote the largest sub-environment of Γ that is linear.
- If for every $x \in \mathrm{dom}(\Gamma)$ we have that $\Gamma(x) = \mathsf{end}$ or $\Gamma(x) = (\mathsf{end}, \mathsf{end})$, we say that Γ is *terminal*.

We define duality of endpoint types in the usual way (note that duality is not defined for base types).

Definition 7 (Duality of endpoint types)**.** Duality of endpoint types is defined inductively by

$$\overline{!T.S} = ?T.\overline{S} \qquad \overline{?T.S} = !T.\overline{S} \qquad \overline{\mathsf{end}} = \mathsf{end}$$

A type $T = (S_1, S_2)$ is *balanced* if $S_1 = \overline{S_2}$. A type environment Γ is balanced if for all $x \in \mathrm{dom}(\Gamma)$ we have that $\Gamma(x)$ is a balanced type or a base type B.

Definition 8 (Depth of types)**.** The depth of an endpoint type S is denoted $d(S)$ and is defined inductively by

$$d(!T.S) = 1 + d(S) \qquad d(?T.S) = 1 + d(S) \qquad d(\mathsf{end}) = 0$$

For a type $T = (S_1, S_2)$ we let $d(T) = \max(S_1, S_2)$. For all other T, we define $d(T) = 0$.

Definition 9 (Addition of type environments)**.** Let Γ_1 and Γ_2 be type environments such that $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$. Then $\Gamma_1 + \Gamma_2$ is the type environment Γ that satisfies

$$\Gamma(x) = \begin{cases} \Gamma_1(x) & \text{if } x \in \mathrm{dom}(\Gamma_1) \setminus \mathrm{dom}(\Gamma_2) \\ \Gamma_2(x) & \text{if } x \in \mathrm{dom}(\Gamma_2) \setminus \mathrm{dom}(\Gamma_1) \end{cases}$$

4.2 Recursion and recursion environments

In our type system, recursion variables are typed with type environments. A *recursion environment* Δ is a function that to each recursion variable X assigns a type environment Γ. The idea is that Γ will represent the names and associated types needed to type a process $\mu X.P$.

Definition 10. A recursion environment Δ is a partial function $\Delta : \mathscr{R} \rightharpoonup (\mathscr{N}_{\text{pol}} \rightharpoonup \mathscr{T})$ with finite support. We let $\Delta_\emptyset$ denote the empty recursion environment.

Definition 11. Let Δ_1 and Δ_2 be recursion environments where for all $X \in \text{dom}(\Delta_1) \cap \text{dom}(\Delta_2)$ we have $\Delta_1(X) = \Delta_2(X)$. $\Delta_1 + \Delta_2$ is the recursion environment Δ satisfying

$$\Delta(X) = \begin{cases} \Delta_1(X) & \text{if } X \in \text{dom}(\Delta_1) \setminus \text{dom}(\Delta_2) \\ \Delta_2(X) & \text{if } X \in \text{dom}(\Delta_2) \setminus \text{dom}(\Delta_1) \\ \Delta_1(X) & \text{otherwise} \end{cases}$$

4.3 Type rules

The set of valid type judgments is defined by the rules in Table 4. The type rules differ from the rules from standard session type systems in their treatment of recursion in two ways.

The rule (VAR) ensures that a recursion variable X can only be well-typed for Γ and Δ if the type environment Γ_1 associated with X mentions all the names in Γ. Moreover, the rule requires that the linear part of the type environment must be *terminal* and that the linear names present when a recursion variable X is reached include the ones found in the type environment used to type the process $\mu X.P$. Therefore, when a recursion variable is reached and a recursive call is made, the restricted names in the unfolding will be new: the existing sessions have been "used up".

The rule (CHAN) ensures that channels that are not session channels can only be bound within a non-recursive process, as the recursion environment present must be $\Delta_\emptyset$. Therefore, names that are not session names cannot accumulate because of recursive calls and lead to an unbounded restriction depth.

The need for private names to be linear inside a recursive process arises because an unlimited channel can be exploited by a recursive process to introduce unbounded nesting, as the following example from [4] illustrates.

Example 3. Consider the following process that cannot be typed; we therefore leave out type annotations and polarities in its description. Let

$$P = (\nu s)(\nu n)(\nu v)(\nu a)(\overline{s}\langle a\rangle \mid \mu S.(s(x).(\nu b)((\overline{v}\langle b\rangle.\overline{n}\langle x\rangle \mid \overline{s}\langle b\rangle) \mid S)))$$

The process can evolve as follows.

$$P \rightarrow^* (\nu s)(\nu n)(\nu v)(\nu a)(P_1 \mid (\nu b)(\nu b')((\overline{v}\langle b\rangle.\overline{n}\langle a\rangle \mid \overline{v}\langle b'\rangle.\overline{n}\langle b\rangle \mid \overline{s}\langle b'\rangle)))$$

where $P_1 = \mu S.(s(x).(\nu b)((\overline{v}\langle b\rangle.\overline{n}\langle x\rangle \mid \overline{s}\langle b\rangle) \mid S))$ can introduce further nesting since the channel s will, when used together with recursion, be used with an arbitrary number of new names that cannot be eliminated.

Note that the (PAR) rule implies that a process P that can be typed in a linear environment must be width-bounded with bound 2, since every name can then occur in either precisely one or precisely two parallel components.

Delegation of session names is handled by (OUT-1); session channels are linear, so the name y^p cannot appear in the continuation P. A special feature of our type system is that endpoint channels that are no longer usable cannot be delegated. Thus, in the rules (IN-1), (IN-2), and (OUT-1), the object type T_1 must be different from end.

(IN-1) $$\frac{\Gamma, x^p : T_2, y : T_1, \Delta \vdash P}{\Gamma, x^p : ?T_1.T_2, \Delta \vdash x^p(y).P}$$

where $T_1 \neq \mathsf{end}$

(IN-2) $$\frac{\Gamma, x^p : \mathsf{Ch}(T_1), y : T_1, \Delta \vdash P}{\Gamma, x^p : \mathsf{Ch}(T_1), \Delta \vdash x(y).P}$$

where $T_1 \neq \mathsf{end}$

(OUT-1) $$\frac{\Gamma, x^p : T_2, \Delta \vdash P}{\Gamma, x^p : !T_1.T_2, y^q : T_1, \Delta \vdash \overline{x^p}\langle y^q \rangle.P}$$

$T_1 \neq \mathsf{end}$

(PAR) $$\frac{\Gamma_1, \Delta_1 \vdash P_1 \quad \Gamma_2, \Delta_2 \vdash P_2}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2 \vdash P_1 \mid P_2}$$

(OUT-2) $$\frac{\Gamma, x : \mathsf{Ch}(T_2), y^q : T_2, \Delta \vdash P}{\Gamma, x : \mathsf{Ch}(T_2), y^q : T_2, \Delta \vdash \overline{x}\langle y^q \rangle.P}$$

where T_2 unlimited

(SESSION) $$\frac{\Gamma, x^+ : S, x^- : \overline{S}, \Delta \vdash P}{\Gamma, \Delta \vdash (\nu x : (S, \overline{S}))P}$$

(NIL) $\Gamma, \Delta \vdash \mathbf{0}$ $\quad$ Γ unlimited

(VAR) $\Gamma, \Delta \vdash X$ $\quad$ $\Delta(X) = \Gamma_1$, $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Gamma_1)$, Γ_{lin} is terminal

(REC) $$\frac{\Gamma, \Delta, X : \Gamma \vdash P}{\Gamma, \Delta \vdash \mu X.P}$$

(CHAN) $$\frac{\Gamma, x : \mathsf{Ch}(T), \Delta_\emptyset \vdash P}{\Gamma, \Delta_\emptyset \vdash (\nu x : \mathsf{Ch}(T))P}$$

Table 4: Type rules for depth-boundedness

5 A subject reduction property

To show our characterization of depth-boundedness, we state a type preservation property: For any well-typed process P, the type of the channel that gives rise to a reduction of P will evolve according to its session type.

Since this channel may be a restricted channel, we must also describe how the session types of restricted channels evolve. Every process in which all bound names are pairwise distinct gives rise to an internal type environment (Definition 12) that collects the types of the bound names; this is an overapproximation of the types of the active names in the process. This environment is defined as follows.

Definition 12. Let P be a process whose bound names are pairwise distinct. Γ_P denotes the internal type environment of P; it is defined by the following clauses (where π denotes a prefix).

$$\Gamma_{P_1|P_2} = \Gamma_{P_1}, \Gamma_{P_2} \qquad \Gamma_{(\nu x:T)P} = x : T, \Gamma_P$$
$$\Gamma_{\mu X.P} = \Gamma_P \qquad \Gamma_{\pi.P} = \Gamma_P$$
$$\Gamma_X = \emptyset$$

The following substitution lemma for variables tells us about the annotated reductions of open process terms.

Lemma 1 (Substitution of variables in reductions)**.** *If* $P[\mu X.P/X] \xrightarrow{\{x\}} P'$ *then* $P \xrightarrow{\{x\}} P''$*, with* $P' = P''[\mu X.P/X]$.

Proof. Induction in the structure of P. □

Lemma 2 (Substitution of variables in typings of recursion)**.** *Suppose* $\Gamma,\Delta \vdash \mu X.P$ *and* $\Gamma,\Delta \vdash Q$*. Then* $\Gamma,\Delta \vdash Q[\mu X.P/X]$.

Proof. Induction in the structure of Q.

$Q = \mathbf{0}$**:** Trivial.

$Q = X$**:** Immediate, since $Q[\mu X.P/X] = \mu X.P$.

$Q = Y$ **(with** $Y \neq X$**):** Immediate.

$Q = Q_1 \mid Q_2$**:** We must then have concluded $\Gamma,\Delta \vdash Q$ using (PAR) with premises $\Gamma_1,\Delta \vdash Q_1$ and $\Gamma_2,\Delta \vdash Q_2$. By induction hypothesis we then have

$$\Gamma_1,\Delta \vdash Q_1[\mu X.P/X]$$
$$\Gamma_2,\Delta \vdash Q_2[\mu X.P/X]$$

We now use the (PAR) rule and get

$$\Gamma,\Delta \vdash Q_1[\mu X.P/X] \mid Q_2[\mu X.P/X]$$

The result now follows by the distributive property of substitution.

$Q = (\nu x : T)P_1$**:** We must have conclude $\Gamma,\Delta \vdash Q$ using (SESSION) with premise $\Gamma,x : S,\Delta \vdash P_1$. By induction hypothesis we have that $\Gamma,x : S,\Delta \vdash P_1[\mu X.P/X]$. But then by the (SESSION) rule we get that $\Gamma,\Delta \vdash (\nu x : T)P_1[\mu X.P/X]$, and we conclude that $\Gamma,\Delta \vdash Q[\mu X.P/X]$.

$Q = \mu Y.Q_1$**:** We must have concluded $\Gamma,\Delta \vdash Q$ using (REC) with premise $\Gamma,\Delta \vdash Q_1$. By induction hypothesis we have

$$\Gamma,\Delta \vdash Q_1[\mu X.P/X]$$

We can now apply (REC) to get the desired result.

$Q = a(x).Q_1$**:** We must have concluded $\Gamma,\Delta \vdash Q$ using (IN) with premise $\Gamma_1,a : T_2,x : T_1,\Delta \vdash Q_1$ and assuming that $\Gamma = \Gamma_1,a :?T_1.T_2$. By applying the induction hypothesis, we get that

$$\Gamma_1,a : T_2,x : T_1,\Delta \vdash Q_1[\mu X.P/X]$$

An application of (IN) and the properties of substitution now gives us the result.

$Q = \overline{a}\langle x\rangle.Q_1$**:** Similar to the previous case.

□

We also need a substitution lemma for names.

Lemma 3 (Substitution of names)**.** *If* $\Gamma,x : T,\Delta \vdash P$ *and* $y \notin n(P)$ *then* $\Gamma,y : T,\Delta \vdash P\{^y/_x\}$.

Proof. Induction in the type rules. □

5.1 A fidelity theorem

For a binary session type system, subject reduction takes the form of *fidelity*: the communications in a well-typed process proceed according to the protocol specified by the channels involved.

Lemma 4 (Subject congruence and normalization)**.** *Suppose* $\Gamma, \Delta \vdash P$. *Then*

- *If* $P \equiv Q$, *then also* $\Gamma, \Delta \vdash Q$
- *If* $P \succ Q$, *then also* $\Gamma, \Delta \vdash Q$

Proof. Induction in the rules defining $\equiv$ and $\succ$. □

The fidelity theorem is a type preservation result: It states that the endpoint types evolve according to the reduction performed. If the name x giving rise to the reduction is free, the annotation of x in the type environment changes. If x is bound, its annotation in the restriction $(\nu x : T)$ changes to $(\nu x : T')$, where $T' = T \downarrow$.

Theorem 5 (Fidelity)**.** *Let* Γ *be a balanced type environment and let* P *be recursion-closed. If* $\Gamma, \Delta_\emptyset \vdash P$ *and* $P \xrightarrow{\alpha} P'$ *where* $x = \mathrm{n}(\alpha)$ *then*

- *if* $x \in \mathrm{fn}(P)$ *and* $\Gamma = \Gamma'', x : T$, *then* $\Gamma', \Delta_\emptyset \vdash P'$ *where* Γ' *is balanced and* $\Gamma' = \Gamma'', x : T \downarrow$
- *if* $x \notin \mathrm{fn}(P)$, *then* $\Gamma, \Delta_\emptyset \vdash P'$ *and if* $\Gamma_P = \Gamma'', x : T$ *then* $\Gamma_{P'} = \Gamma'', x : T \downarrow$ *and* $\Gamma_{P'}$ *is balanced.*

Proof. Induction in the reduction rules.

Com-Annot Here, only the first case is relevant. We know that $P = a^p(x).P_1 \mid \overline{a^{\overline{p}}}\langle y^q \rangle.P_2$. Since $\Gamma, \Delta_\emptyset \vdash P$, we must have that $\Gamma = \Gamma_1 + \Gamma_2$ where

$$\Gamma_1, \Delta_\emptyset \vdash a^p(x).P_1 \tag{1}$$

and

$$\Gamma_2, \Delta_\emptyset \vdash \overline{a^{\overline{p}}}\langle y^q \rangle.P_2. \tag{2}$$

We must have used (IN) to conclude (1), so we have $\Gamma_1(a^p) = ?T_1.S$ and, letting $\Gamma_1 = \Gamma_1' + a^p : ?T_1.S$, we have

$$\Gamma_1', a^p : S, x : T_1, \Delta_\emptyset \vdash P_1. \tag{3}$$

Similarly, we must have used (OUT) to conclude (2). Since Γ is balanced, we have $\Gamma_2(a^{\overline{p}}) = !T_1.\overline{S}$. By the substitution lemma Lemma 3 and (3), we have $\Gamma_1', a^p : S, y^q : T_1, \Delta_\emptyset \vdash P_1\{^y/_x\}$. Similarly, letting $\Gamma_2 = \Gamma_2', a^{\overline{p}} : !T_1.\overline{S}, y^q : T_1$, we get $\Gamma_2', a^{\overline{p}} : \overline{S}, \Delta_\emptyset \vdash P_2$. An application of (PAR) now gives us that

$$\Gamma_1' + \Gamma_2' + a^p : S, a^{\overline{p}} : \overline{S}, y : T_1, \Delta_\emptyset \vdash P_1\{^y/_x\} \mid P_2$$

The type environment $\Gamma_1' + \Gamma_2' + a^p : S, a^{\overline{p}} : \overline{S}, y : T_1$ is balanced, since Γ_1' and Γ_2' are balanced and since y must appear with polarity $\overline{q}$ in one of these (because Γ is balanced).

Par-Annot Since $\Gamma, \Delta_\emptyset \vdash P \mid Q$, we have that $\Gamma_1, \Delta_\emptyset \vdash P$ where $\Gamma = \Gamma_1 + \Gamma_2$. The result now follows easily by an application of the induction hypothesis to the reduction $P \xrightarrow{a} P'$ and subsequent use of the (PAR) rule.

New-Annot There are two cases here: whether $x = a$ or $x \neq a$. In both cases, the result follows immediately by the induction hypothesis and use of the (SESSION) rule.

Unfold-Annot Follows from Lemma 4 and a direct application of the induction hypothesis.

Struct-Annot Follows from Lemma 4 and a direct application of the induction hypothesis.

□

6 Soundness of the type system for depth-boundedness

In the following we will consider the correctness properties of the type system for depth boundedness.

6.1 Properties of unfolding and nesting

We first establish a collection of properties that hold for arbitrary processes. Next we show that there are further properties guaranteed by well-typed processes.

The following lemma describes how reductions occur. Reductions can happen directly or may need unfoldings.

Lemma 6. *Let P be an arbitrary recursion-closed process.*

1. *If $P \xrightarrow{\{x\}} P'$, then there exists an unfolding context C and a process Q such that $P \equiv C[Q]$ and $P' \equiv C[Q']$, and $Q \xrightarrow{\{x\}} Q'$ is an instance of* (COM-ANNOT).
2. *If $P \xrightarrow{\{\mathsf{rec},x\}} P'$ then there exists an unfolding context C and either $P \equiv C[\mu X.Q_1]$ for some Q_1 where $Q_1[\mu X.Q_1/X] \xrightarrow{\{x\}} Q_1'$ and $P' \equiv C[Q_1']$ or $P \equiv C[(\mu X.Q_1) \mid Q_2]$ where $Q_1[\mu X.Q_1/X] \mid Q_2 \xrightarrow{\{\mathsf{rec},x\}} Q_1' \mid Q_2'$ is an instance of* (COM-ANNOT) *and $P' \equiv C[Q_1' \mid Q_2']$.*

Proof. By induction in the annotated reduction rules. The proof of Case 2 uses Case 1. □

6.2 Nesting properties of well-typed processes

We now restrict our attention to well-typed processes. The only potential source of unbounded restriction depth is the presence of recursion, and we now show how our type system controls the introduction of new bound names in the presence of recursion.

The first lemma tells us that bound names introduced by an unfolding do not interfere with names in its surrounding process that represent terminated channels.

Lemma 7. *If $\Gamma,\Delta \vdash (\nu c : (\mathsf{end},\mathsf{end}))P$ then $c \notin \mathrm{fn}(P)$.*

The following lemma tells us that names that appear in an unfolding context will not reappear free in the result of unfolding a recursive process.

Definition 13 (Known bound names)**.** The set of known bound names in an unfolding context is defined by

$$\mathrm{kn}([] \mid P) = \emptyset \qquad\qquad \mathrm{kn}((\nu x : T)C) = \{x\} \cup \mathrm{kn}(C)$$

Lemma 8. *Suppose we have $\Gamma,\Delta \vdash C[X]$ where $C[X]$ is recursion-closed and X occurs in $\mu X.P$. Then we also have $\Gamma,\Delta \vdash C[\mu X.P]$ and $\mathrm{kn}(C) \cap \mathrm{fn}(\mu X.P) = \emptyset$.*

Theorem 9. *Let P be recursion-closed. Suppose $\Gamma,\Delta_\emptyset \vdash P$. Then P is depth-bounded.*

Outline. The session types provide a bound on the nesting depth of a well-typed process. Suppose $\Gamma,\Delta_\emptyset \vdash P$. Let $d(\Gamma,P)$ denote the sum of the depths of the session types in Γ and in Γ_P, i.e.

$$d(\Gamma,P) = \sum_{x:T\in\Gamma \text{ or } x:T\in\Gamma_P} d(T)$$

In a process P with k bound names, we know from Theorem 5 that there can be at most $(d(\Gamma+\Gamma_P)/2)-k$ reduction steps before an unfolding has to take place, since every reduction step will decrease the depth

of one of the session types in $\mathrm{ran}(\Gamma) \cup \mathrm{ran}(\Gamma_P)$. Whenever unfoldings occur, the bound names in the unfolding are distinct from those already known and will all be names of session channels. Moreover, when the unfolding is reached, the channel used in the reduction will no longer be available. As a consequence we see that the nesting depth will therefore not increase. □

7 A type system for name-boundedness

We now show to modify our previous type system such that every well-typed process will be name-bounded. The challenge is again one of controlling recursion. As before, the crucial observation is that if private channels are linear, then all the channels that have been used when a recursion unfolding takes place, can then be discarded.

In the case of name-boundedness, extra care must be taken, since recursion may now accumulate an unbounded number of finite components that each contain pairwise distinct bound names.

Example 4. The untyped process

$$P_2 = \mu X.(\nu r_1)(\nu r_2)(\overline{r_1}\langle a\rangle.X \mid r_1(x).X \mid \overline{r_2}\langle a\rangle \mid r_2(x))$$

shows two problems that must be dealt with. Firstly, unfolding a recursion may introduce more parallel recursive components that each have their own bound names. In this case, every communication on r_1 will introduce two new parallel copies of the recursive process. Secondly, unfolding may introduce finite (non-recursive) components which contain bound names that persist – in this case, we get new copies of $(\nu r_2)(\overline{r_2}\langle a\rangle \mid r_2(x))$ for every unfolding.

The type language is

$$S_{\mathsf{lin}} ::= ?T_{\mathsf{lin}}.S_{\mathsf{lin}} \mid !T_{\mathsf{lin}}.S_{\mathsf{lin}} \mid \mathsf{end} \qquad S_{\mathsf{un}} ::= \mathsf{Ch}(S_{\mathsf{un}})$$
$$T_{\mathsf{lin}} ::= (S_{\mathsf{lin}}, S_{\mathsf{un}}) \mid (S_{\mathsf{lin}}, S_{\mathsf{lin}}) \qquad T ::= T_{\mathsf{lin}} \mid S_{\mathsf{un}}$$

Note that names of unlimited type S_{un} can only be used to delegate channels of unlimited type.

The type rules are as in the original type system, but we now modify the notions of addition for type environments and for recursion environments. We add pairs (Γ_1, Δ_1) and (Γ_2, Δ_2) as follows.

Definition 14. Let Γ_1, Γ_2 be type environments and let Δ_1, Δ_2 be recursion environments where at least one of Δ_1, Δ_2 is $\Delta_\emptyset$. We define $(\Gamma_1, \Delta_1) + (\Gamma_2, \Delta_2) = (\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2)$ where Γ_1 is unlimited if $\Delta_1 = \Delta_\emptyset$ and Γ_2 is linear if $\Delta_2 \neq \Delta_\emptyset$.

The intention is that an empty recursion environment must now go together with an unlimited type environment. In other words: Non-recursive subprocesses can only contain unlimited names.

We say that a type environment Γ is *limited* if for every $x \in \mathrm{dom}(\Gamma)$ we have that $\Gamma(x) = (T_{\mathsf{lin}}, \overline{T_{\mathsf{lin}}})$ for some T_{lin}. That is, the environment is balanced, and no name has an unlimited type.

A type environment Γ is *skew* if $\Gamma = \Gamma_1 + \Gamma_2$ with $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$, Γ_1 is linear and for all $x \in \mathrm{dom}(\Gamma_2)$ we have that $\Gamma(x) = (T_{\mathsf{lin}}, T_{\mathsf{un}})$ for some $T_{\mathsf{lin}}, T_{\mathsf{un}}$.

7.1 Fidelity

As in the case of the previous type system, we need a fidelity result.

Theorem 10 (Fidelity)**.** *Let Γ be a type environment. If $\Gamma, \Delta_\emptyset \vdash P$ and $P \xrightarrow{x} P'$ then*

- *if $x \in \mathrm{fn}(P)$ and $\Gamma = \Gamma'', x : T$, then $\Gamma', \Delta_\emptyset \vdash P'$ where Γ' is balanced and $\Gamma' = \Gamma'', x : T \downarrow$*
- *if $x \notin \mathrm{fn}(P)$, then $\Gamma, \Delta_\emptyset \vdash P'$ and if $\Gamma_P = \Gamma'', x : T$ then $\Gamma_{P'} = \Gamma'', x : T \downarrow$ and $\Gamma_{P'}$ is balanced.*

Since the new type system specialized the previous one, this result is easily established.

7.2 Soundness for name-boundedness

We will show that if a process is well-typed in a limited environment, then it is name-bounded.

To show that a well-typed process P is name-bounded, we will show that

- For some k, whenever $P \rightarrow^* P'$, then P' has at most k recursion instances in P'
- For some m, whenever $P \rightarrow^* P'$, every recursive subprocess of P' contains at most m distinct bound names
- There are only free names in the non-recursive part of P

Since every well-typed process is known to be depth-bounded, the result will then follow.

Our first lemma gives a characterization of well-typed recursive processes: They can contain at most one instance of each recursion variable.

Lemma 11. *Let $\mu X.P$ be a process for which all binding occurrences of recursion variables are distinct. If $\Gamma, \Delta \vdash \mu X.P$, there is at most one occurrence of X in P.*

Proof. Suppose to the contrary that there is more than one occurrence of X in P. We then have that $\mu X.P = \mu X.(\nu\vec{n})(C_1[X] \mid C_2[X] \mid P')$ where n is a set of names (possibly empty), and C_1 and C_2 are process contexts.

The derivation of the type judgement $\Gamma, \Delta \vdash \mu X.(\nu\vec{n})(C_1[X] \mid C_2[X] \mid P')$ must have used the (REC) type rule in its final step, having premise $\Gamma, \Delta, X : \Gamma \vdash (\nu\vec{n})(C_1[X] \mid C_2[X] \mid P'$. But the derivation of this judgement must have used the (SESSION) rule a number of times, preceded by an application of (PAR) with premises $\Gamma_1, \Delta, X : \Gamma \vdash C_1[X]$ and $\Gamma_2, \Delta_\emptyset \vdash C_2[X]$ where Γ_2 is unlimited. However, there can be no derivation of the latter, since this would require the rule (VAR) in which it is assumed that the type environment is linear.

We therefore conclude that our initial assumption was wrong; there can be at most one occurrence of X in P. □

This lemma tells us that there can be no finite, non-recursive subprocesses of a recursive process with their own bound names; any bound name found in a non-recursive subprocess will also appear in the recursive part of the process.

Lemma 12. *If $\Gamma, \Delta \vdash \mu X.(C[X] \mid P)$ where $\mu X.(C[X] \mid P)$ is in inner normal form and $C[X]$ is a process context, then for every $n \in \mathrm{bn}(P)$ we have that $n \in \mathrm{bn}(C[X])$.*

Proof. Consider a name $n \in \mathrm{bn}(P)$. Suppose $n \notin \mathrm{bn}(C[X])$. Since $\mu X.(C[X] \mid P)$ is in inner normal form, we would then have a subprocess $(\nu n : T)P'$ of P that would be typed using the (SESSION) rule. But for this rule to be applicable, a recursion variable must be present in the type environment. This cannot be the case, as P is non-recursive. □

We now show that the number of recursive subprocesses that will appear in any reduction sequence for a well-typed process is bounded. Let $\mathsf{recs}(P)$ denote the number of simultaneous recursion instances in P and let $\mathsf{recv}(P)$ denote the multiset of recursion variable occurrences in P.

Together, the following two lemmas give an upper bound on the number of recursion instances in any reduction sequence of a well-typed process.

Lemma 13. *Suppose $\Gamma, \Delta \vdash P$ and $P \xrightarrow{\alpha} P'$ was proved without using instances of* (UNFOLD-ANNOT). *Then $\mathsf{recs}(P) \geq \mathsf{recs}(P')$.*

Lemma 14. *Suppose* $\Gamma, \Delta \vdash P$ *where* $\mathrm{dom}(\Delta) \cap \mathsf{recv}(P) = \emptyset$ *and* $P > P_1$. *Then* $\mathsf{recs}(P) \geq \mathsf{recs}(P_1)$.

The following normal form theorem is crucial.

Theorem 15. *If* $\Gamma, \Delta \vdash P$, *then there exists a* $k \geq 0$ *such that whenever* $P \to^* P'$, *we have* $P \equiv P_1 \mid P_2$ *where* $\mathsf{recs}(P_1) \leq k$, $\mathsf{recs}(P_2) = 0$ *and* P_2 *contains no restrictions.*

Proof. We show that for all $n \geq 0$, if $P \to^n P'$, then we have $P \equiv P_1 \mid P_2$ where $\mathsf{recs}(P_1) \leq k$, $\mathsf{recs}(P_2) = 0$ and P_2 contains no restrictions. The proof of this proceeds by induction in n.

$n = 0$**:** Here we let $k = \mathsf{recs}(P)$ and proceed by induction in the type derivation of $\Gamma, \Delta \vdash P$. We consider each rule in turn.

(IN-1), (IN-2), (OUT-1) and (OUT-2): None of these rules could have been used, since P would then have no reductions.

(PAR): Here we can use the commutativity and associativity axioms for structural congruence to rewrite P in the desired form.

(VAR): Cannot apply, since we assume that $\Gamma, \Delta \vdash P$.

(REC), (NIL), (SESSION): These are immediate.

Assume for n**, prove for** $n+1$**:** This is a straightforward induction in the type rules.

□

Theorem 16. *If* $\Gamma, \Delta \vdash P$, *then* P *is name-bounded.*

Proof. There is a $k \geq 0$ such that if $\Gamma, \Delta \vdash P$, whenever $P \to^* P'$, there are at most k recursive subprocesses of P'. Since the new type system is a subsystem of the type system for depth-boundedness, there exists a d such that the recursion depth of P' is at most d for any such P'.

Every bound name in a non-recursive subterm of a recursive subprocess occurs in the recursion part as well. Now consider an outer normal form P'' of P'. We have $P'' = (\nu x_1) \ldots (\nu x_d) P^{(3)}$ for some $P^{(3)}$ that does not contain restrictions at the outermost level. Moreover, for some $k' \leq k$ we have $P^{(3)} \equiv P_1^{(3)} \mid \cdots P_{k'}^{(3)} \mid P_{k'+1}^{(3)}$ where $P_1^{(3)}, \ldots, P_{k'}^{(3)}$ contain recursion instances and $P_{k'+1}^{(3)}$ is a process not containing recursion instances. We know that for some d there are at most $d \cdot k$ bound names in P'. □

8 The relation to other classes of processes

Because of the use of binary session types, typable process in our systems will be width-bounded with name width 2. On the other hand, both type systems allow us to type processes that are not finitary. The classes of typable processes differ from those already studied. The process $P_1 \stackrel{\text{def}}{=} (\mu X.(\nu a)a(x).X \mid \overline{a}\langle b\rangle \mid \overline{b}\langle c\rangle)$ is not a finite-control process, since the reduction sequence $P \to^k P_1 \mid \overline{b}\langle c\rangle \mid \cdots \mid \overline{b}\langle b\rangle$ that results in $k-1$ parallel components, each being a simple output, shows that the number of parallel components along a computation can be unbounded for a well-typed process. This means that P_1 is neither a finite-control process [3] nor a bounded process in the sense of [2]. On the other hand, P_1 is depth-bounded, and in fact also width-bounded as every bound name occurs in precisely two parallel components. Moreover, the typable processes are incomparable with the processes studied in [1] since these do not allow for delegation of input capabilities.

9 Conclusions and ideas for further work

In this paper we have presented two session type systems for a π-calculus with recursion. One guarantees depth-boundedness, and the other system, which is a subsystem of it, guarantees name-boundedness. Both systems assume that names are always used in finite-length sessions before a recursive call is initiated.

In the paper by D'Osualdo and Ong [4] a type inference algorithm is proposed that makes it possible to provide a safe bound on the restriction depth for depth-bounded processes. A further topic of investigation is to adapt the type inference algorithm proposed in [6] to the setting of the type systems of the present paper. We conjecture that this is straightforward. The type systems presented in this paper are simpler than many other session type systems, in that they do not involve recursive types; the sole difference is that of the presence of recursion instead of replication in the π-calculus.

In both systems, the number of parallel components in a well-typed system can be unbounded, and well-typed processes need not be finite-control. Conversely, finite-control processes need not be well-typed in the present systems, since finite-control processes are not necessarily width-bounded with width 2.

Another important question to be answered is that of the exact relationshop between our type system for depth-boundedness and the type system due to D'Osualdo and Ong [4].

References

[1] Roberto M. Amadio and Charles Meyssonnier. On decidability of the control reachability problem in the asynchronous pi-calculus. *Nordic J. of Computing*, 9(2):70–101, June 2002.

[2] Luís Caires. Behavioral and spatial observations in a logic for the pi-calculus. In Igor Walukiewicz, editor, *Proceedings of FOSSACS 2004*, LNCS 2987, pp. 72-89, Springer, 2004. doi:10.1007/978-3-540-24727-2_7.

[3] Mads Dam. Model checking mobile processes. *Inf. Comput.*, 129(1):35–51, 1996. doi:10.1006/inco.1996.0072.

[4] Emanuele D'Osualdo and Luke Ong. A type system for proving depth boundedness in the pi-calculus. *CoRR*, abs/1502.00944, 2015. `http://arxiv.org/abs/1502.00944`

[5] Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2-3):191–225, 2005. doi:10.1007/s00236-005-0177-z.

[6] Eva Fajstrup Graversen, Jacob Buchreitz Harbo, Hans Hüttel, Mathias Ormstrup Bjerregaard, Niels Sonnich Poulsen, and Sebastian Wahl. Type inference for session types in the π-calculus. In T. Hildebrandt, A. Ravara, J. van der Werf, and M. Weidlich (ed.) *Proc. of WS-FM 2014 and WS-FM/BEAT 2015*, LNCS 9421, Springer, 2015. doi:10.1007/978-3-319-33612-1_7.

[7] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of ESOP'98*, pages 122–138, 1998. doi:10.1007/BFb0053567.

[8] Rainer Hüchting, Rupak Majumdar, and Roland Meyer. *A Theory of Name Boundedness*, pages 182–196. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-40184-8_14.

[9] Ivan Lanese, Jorge A. Pérez, Davide Sangiorgi, and Alan Schmitt. On the expressiveness and decidability of higher-order process calculi. *Inf. Comput.*, 209(2):198–226, 2011. doi:10.1016/j.ic.2010.10.001.

[10] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off Go: Liveness and safety for channel-based programming. In *POPL 2017*, pp. 748–761, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.

[11] Roland Meyer. On boundedness in depth in the pi-calculus. In Giorgio Ausiello, Juhani Karhumäki, Giancarlo Mauri, and C.-H. Luke Ong, editors, *Proceedings of TCS 2008*, volume 273 of *IFIP*, pages 477–489. Springer, 2008. doi:10.1007/978-0-387-09680-3_32.