Using session types for reasoning about boundedness in the $\pi$-calculus

Hans Hüttel

Received: 31 March 2018 / Accepted: 23 August 2019
© Springer-Verlag GmbH Germany, part of Springer Nature 2019

Abstract
The $\pi$-calculus is a well-established theoretical framework for describing mobile and parallel computation using name passing, and a central notion here is that of name binding. Unfortunately, non-trivial properties of $\pi$-calculus processes such as termination and bisimilarity are undecidable as a consequence of the fact that the calculus is Turing-powerful. The classes of depth-bounded and name-bounded processes are classes of $\pi$-calculus processes that impose constraints on how name binding is used in a process. A consequence of this is that some of the important decision problems that are undecidable for the full calculus now become decidable. However, membership of these classes of processes is undecidable, so it is difficult to make use of the positive decidability results in practice. In this paper we use binary session types to devise two type systems that give a sound and decidable characterization of each of these two 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

The $\pi$-calculus [14,15] is intended as a simple theoretical framework for understanding the properties of mobile and parallel computation through the notion of name passing. In this setting the notion of name restriction becomes particularly important, and 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, private name, known only to $P$. It is the interplay between restriction and replication (or recursion) that leads to the $\pi$-calculus being Turing-powerful. Without either of these two constructs, this is no longer the case [10].
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?”. These problems are important from a practical perspective, as they are important examples of liveness properties of programs that have been the focus of much attention in program verification [16] since the early 1980s.

Several classes of processes have been identified for which (some of) these problems remain decidable. Examples are the finitary processes, that is, processes without replication or recursion, the finite-control processes [4] in which every process has a uniform bound on the number of parallel components in any computation, the bounded processes [3] 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 [12] and is the most expressive known fragment of the \( \pi \)-calculus for which interesting verification problems remain decidable [17].

The class of depth-bounded processes is expressive: A depth-bounded process can have infinitely many states and contain infinitely many names bound by restriction, but the restriction depth is uniformly bounded. Moreover, the class contains a variety of other decidable subsets of the \( \pi \)-calculus, including the classes of finitary and finite-control processes mentioned above.

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 original proof of this uses the notion of well-structured transition systems to obtain a backward algorithm. Wies et al. have since proposed forward algorithms for the coverability problem [17]; the exact complexity of the problem is an open problem.

Depth-boundedness is in itself an interesting program property from the point of view of the analysis of how data structures evolve during the execution of a program. It is well-known that the \( \pi \)-calculus can be used to express data structures using the notion of name scoping, and it follows that the size of data structures in a depth-bounded process will then itself be bounded. This means that a depth-bounded process cannot have e.g. queues that grow indefinitely. 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 [12].

In a more recent paper [5], 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 \( \pi \)-calculus processes is that of name-bounded processes, introduced by Hüchting et al. [9]. 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 private names. Name-bounded processes are depth-bounded, so all positive decidability results carry over to this setting. However, some problems that are undecidable for depth-bounded processes, notably the reachability problem [13], now also become decidable [9].

The goal of this paper is to use binary session types [8] to give sound characterizations of depth-boundedness, respectively name-boundedness in the \( \pi \)-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 [5] 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; in our type system we can use this to ensure that new private names are only introduced whenever existing restricted 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, we make use of the fact that the type system is a more restrictive version of that for depth-boundedness.

The rest of our paper is organized as follows. Section 2 describes the version of the $\pi$-calculus that we will consider in this paper; Sect. 3 introduces the notions of boundedness. Section 4 presents a type system for depth-bounded processes, which is analyzed in Sects. 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 $\pi$-calculus with recursion

The $\pi$-calculus is a model of message-passing concurrent computation introduced by Milner et al. [14,15] for which the notion of name is essential and subsumes the usual notions of channels, variables and simple data. Names in the $\pi$-calculus can be scoped using restriction. In a process $(\nu x)P$ the name $x$ is private within $P$. Names can be communicated across named channels using the two forms of prefixing in the calculus: The input prefix $x(y).P$ receives a name on the channel named $x$ and binds it to $y$ within the continuation $P$. The output prefix $x(z).P$ sends out the name $z$ on the channel named $x$ and continues as $P$. In many presentations of the $\pi$-calculus infinite behaviour is made possible by the syntactic construct of replication: $!P$ denotes a process consisting of infinitely many parallel components, each of which is $P$. Replication makes it possible to express the usual notion of recursion using the name-passing and scoping mechanisms.

However, in this presentation we follow Meyer [12] and use a $\pi$-calculus with recursion instead of replication. The reason behind this choice of syntax is that we would like infinite behaviours to make use of private 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, as it introduces infinitely many distinct names.

### 2.1 Syntax

We consider a typed version of the $\pi$-calculus, in which private names are annotated with types. We assume the existence of a countably infinite set of names, $\mathcal{N}$, and let $a, b, \ldots$ and $x, y, \ldots$ range over $\mathcal{N}$. Moreover, we assume a countably infinite set of recursion variables, $\mathcal{R}$, and let $X, Y, \ldots$ range over $\mathcal{R}$.
2.1.1 Processes and types

Processes are annotated with types, so we first introduce our set of types. The type system is a non-recursive variant of the binary session types introduced by Gay and Hole [6], and in this paper we follow the presentation of [2].

Types The set of types is called $T$ and is defined by the formation rules

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

In a session type discipline, named channels are seen as having two endpoints belonging to different parallel components. A type $T$ can therefore be a linear endpoint type $S$ or pair of endpoints $(S_1, S_2)$, or an unlimited channel type $\text{Ch}(B)$, base type $B$ or one of the terminated types end and (end, end). Base type are the types of simple data such as numbers, truth-values and the singleton type that admits only the unit value ($\text{.}$). We write $\text{lin} T$ to denote that $T$ is linear and $\text{un} T$ to denote that $T$ is unlimited.

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 $\text{end}$ is the type of an endpoint that allows no further communication. If $T = (! T_1, S_2, ? T'_1, S'_2)$ we let $T \downarrow = (S_1, S'_2)$; this denotes the successor of a pair of endpoint types. If $T = \text{Ch}(T_1)$, then $T \downarrow = T$.

Processes Following [6] we will use a version of the $\pi$-calculus with polarized names in order to distinguish between the two endpoints of a channel. We assume polarities ranked over by $p, q, \ldots$. The polarities $+$ and $-$ are dual; we define the dual $\overline{p}$ of a polarity $p$ by $\overline{+} = -$ and $\overline{-} = +$. The empty polarity $\varepsilon$ 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 $N_{\text{pol}}$. In some of the examples of processes that follow, we leave out the polarities whenever these are not of importance.

The formation rules of processes are given by

$$
P ::= x^p(y). P_1 \mid \overline{x^q(y^q)}. P_1 \mid P_1 \mid P_2 \mid (v x : T) P_1 \mid \mu X. P_1 \mid X \mid 0
$$

$$
p ::= + \mid - \mid \varepsilon
$$

As before, $x^p(y). P_1$ denotes a process that inputs a name on channel $x$ and continues as $P_1$; the unpolarized name $y$ is bound in $P_1$. $\overline{x^q(y^q)}. P_1$ is a process that outputs the polarized name $y^q$ on channel $x$ and continues as $P_1$. The process $P_1 \mid P_2$ is the parallel execution of $P_1$ and $P_2$. The process $0$ is the inactive, terminated process.

$\mu X. P_1$ is a recursive process with body $P_1$ and recursion variable $X$; this $X$ is called a binding occurrence. We assume that every recursive process is guarded; each occurrence of a recursion variable $X$ must be found underneath an input or an output prefix. A recursion variable $X$ is free in a process $P$ if it does not have a binding occurrence in $P$. The set of free recursion variables in a process $P$ is denoted $\text{frec}(P)$.

Moreover, we always assume that for any process $P$, if $\mu X. P_1$ and $\mu Y. P_2$ are subexpressions of $P$, then $X$ and $Y$ are distinct.

If $P$ and $Q$ are processes, we let $P[Q/x]$ denote the process obtained by substituting all free occurrences of $X$ in $P$ by $Q$. We are particularly interested in unfoldings of recursive processes: For any process $\mu X. P$, its unfolding is the process $P[\mu X. P/x]$. 

 Springer
A process $P$ is **recursion-closed** if $\text{frec}(P) = \emptyset$. For a process $P$, we will refer to any subprocess $\mu X. P_1$ as a **recursion instance**.

**Session channels** In the restriction $(\nu x : T) P_1$ the unpolarized name $x$ is bound in $P_1$ and annotated with type $T$, where $T \in T$, the set of types. 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 private names evolve.

**Example 1** Consider the process $P$ given by

$$
(\nu c : (T, \overline{T}))(\overline{a}(c).c^+(x).c^-(b) | a(y).\overline{a}(i).y(x).Q(x))
$$

where $Q(x)$ is an unspecified successor process that depends on the free name $x$. $P$ is a process with two parallel components, $P_1$ and $P_2$. In $P_1$, we first output the private name $c$ on the channel $a$. In $P_2$, we receive a name on channel $a$ and bind $y$ to it; this will lead to $c$ being bound to $y$, so $c$ is now known to both parallel components. We assume that the free name $i$ has type $\text{Int}$ and that the free name $b$ has type $\text{Bool}$, where $\text{Int}$ and $\text{Bool}$ are base types.

The private channel name $c$ has the session type $(T, \overline{T})$, which we now describe. Assume for this example that we have the base types $\text{Int}$ of integers and $\text{Bool}$ of booleans. In $P$, the channel named $c$ has two **endpoints**, $c^+$ and $c^-$. In $P_1$ the endpoint $c^+$ follows the protocol $T$:

$$
c^+ : ?\text{Int}!\text{Bool} . \text{end}
$$

and in $P_2$ the endpoint $c^-$ follows the dual protocol $\overline{T}$:

$$
c^- : !\text{Int} ?\text{Bool} . \text{end}
$$

**Properties of names** The sets of free and bound names of a process, $\text{fn}(P)$ and $\text{bn}(P)$, are defined as usual.

For free names, we have

$$
\begin{align*}
\text{fn}(x^P(y).P_1) &= \{x\} \cup (\text{fn}(P_1) \setminus \{y\}) \\
\text{fn}(\overline{x^P}(y^d).P_1) &= \{x, y\} \cup \text{fn}(P_1)
\end{align*}
$$

$$
\begin{align*}
\text{fn}(P_1 | P_2) &= \text{fn}(P_1) \cup \text{fn}(P_2) \\
\text{fn}((\nu x : T) P_1) &= \text{fn}(P_1) \setminus \{x\} \\
\text{fn}(\mu X. P_1) &= \text{fn}(P_1) \\
\text{fn}(X) &= \text{fn}(0) = \emptyset
\end{align*}
$$

and for bound names, we have

$$
\begin{align*}
\text{bn}(x^P(y).P_1) &= \{y\} \cup \text{bn}(P_1) \\
\text{bn}(\overline{x^P}(y^d).P_1) &= \text{bn}(P_1)
\end{align*}
$$

$$
\begin{align*}
\text{bn}(P_1 | P_2) &= \text{bn}(P_1) \cup \text{bn}(P_2) \\
\text{bn}((\nu x : T) P_1) &= \text{bn}(P_1) \cup \{x\} \\
\text{bn}(\mu X. P_1) &= \text{bn}(P_1) \\
\text{bn}(X) &= \text{fn}(0) = \emptyset
\end{align*}
$$

The subset of the bound names in $P$ that are private names, that is, names bound by some restriction is denoted by $\text{pn}(P)$, and the set of all names in $P$ is denoted by $\text{n}(P)$. 

---

Using session types for reasoning about boundedness...
Table 1 Structural congruence: axioms and rules

| Axiom   | Expression                                      |
|---------|------------------------------------------------|
| (NEW-1) | \((\nu x : T)(\nu y : T')P \equiv (\nu y : T')(\nu x : T)P\) |
| (NEW-2) | \((\nu x : T)P \mid Q \equiv (\nu x : T)(P \mid Q)\) if \(x \notin fn(Q)\) |
| (PAR-1) | \(P \mid Q \equiv Q \mid P\) |
| (PAR-2) | \((P \mid Q) \mid R \equiv P \mid (Q \mid R)\) |

We say that processes \(P\) and \(Q\) are \(\alpha\)-convertible, if \(P\) and \(Q\) are syntactically identical up to a renaming of zero or more bound names. To simplify the presentation, we from now on assume that all free and bound names in a process are distinct and that \(\alpha\)-conversions are only used to ensure this distinction.

We let \(P[y/x]\) denote process in which all free occurrences of the name \(x\) in \(P\) have been replaced by the name \(y\), using \(\alpha\)-conversion to avoid name clashes where necessary.

An important notion is that of active names [5].

**Definition 1** A subterm \(P_1\) of a process \(P\) is active if it is does not occur underneath a prefix. A name \(x\) is active when it is bound by an active restriction \((\nu x : T)P_1\).

**Example 2** In the process \((\nu x : T)\bar{y}(z).\bar{z}(x).\bar{w}(x)\), the subterm \(\bar{y}(z).\bar{z}(x).\bar{w}(x)\) is active, whereas \(\bar{z}(x).\bar{w}(x)\) is not. The name \(x\) is active.

### 2.1.2 Structural congruence

Structural congruence is the least congruence relation for the process constructs that is closed under the axioms in Table 1. The axiom (NEW-1) allows us to use a shorthand for successive restrictions; if \(n : T = (n_1 : T_1, \ldots, n_k : T_k)\), then we write \((\nu n : T)P\) for \((\nu n_1 : T_1) \cdots (\nu n_k : T_k)P\).

Following Meyer [12], we sometimes consider processes in normal form. A process is in outer normal form if every restriction not underneath a prefix appears at the outermost level.

**Definition 2** (Normal forms) Let \(P\) be a process. \(P\) is in outer normal form if \(P = (\nu x : T)P_1\) 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 outer normal form.

**Proof** Induction in the structure of \(P\).

The interesting case is that of \(P = P_1 \mid P_2\). If we have an outer normal form \((\nu m : T)P_{11}\) for \(P_1\) and an outer normal form \((\nu n : U)P_{22}\) for \(P_2\), then by (NEW-2), we have that \((\nu n, m : T, U)(P_{11} \mid P_{22})\) is an outer normal form for \(P_1 \mid P_2\). \(\square\)

Note that normal forms are in general not unique due to the commutativity properties of parallel composition and nested restrictions under structural congruence.

### 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 \(\alpha = \{\text{rec}, a\}\) for \(a \in \mathcal{N}\).

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

A reduction annotated with \([a]\) denotes a communication on a channel named \(a\).
Example 2 (continued) Consider again the process $P$ given by

$$(vc : (T, \overline{T}))(\overline{\alpha}(c).c^+(x).c^+(b) | a(y).\overline{y}(i).y(x).Q(x))$$

with the type $(T, \overline{T})$ as previously defined. The annotated reduction semantics will yield the reduction sequence

$$P \xrightarrow{[a]} (vc : (T, \overline{T}))(c^+(x).c^+(b) | \overline{c}(i).c(x).Q(x))$$

$$\xrightarrow{[c]} (vc : (!\text{Bool} \cdot ?\text{Bool}))((c^+(b) | c(x).Q(x))$$

Notice how the session type of $c$ is reduced, when $c$ is used for communication. The annotation containing $\text{rec}$ indicates that recursive unfolding was necessary to obtain the reduction.

Example 3 Consider the process $P_1$ given by

$c(x).0 | \mu X.\overline{c}(d).X$

We would like the semantics to express that $\mu X.\overline{c}(d).X$ unfolds to $\overline{c}(d) \cdot \mu X.\overline{c}(d).X$, and an application of (Com- Annot) to this unfolded process should then give us that

$$P_1 \xrightarrow{\text{rec},c} 0 | \mu X.\overline{c}(d).X$$

Since $0 | \mu X.\overline{c}(d).X \equiv \mu X.\overline{c}(d).X$, we have from (Struct- Annot) that

$$P_1 \xrightarrow{\text{rec},c} \mu X.\overline{c}(d).X$$

The reduction rules are found in Table 2. Note that in (New- Annot) the type associated with the bound name $x$ evolves, if $x$ is responsible for the communication and $T$ is a session type. We define the set of names $n(\alpha)$ in an annotation $\alpha$ by the clauses $n([a]) = a$ and $n([\text{rec}, a]) = a$.

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 whose hole [] indicates the place where a subprocess that participates in a reduction step appears as the direct result of unfolding a recursive process. In the rule (Unfold- Annot) the unfolding relation appears in the premise; by applying (Unfold- Annot) it is possible to express that a recursive process can be unfolded more than once.

Definition 3 (Unfolding contexts) The set of unfolding contexts is given by the formation rules

$$C ::= [] | P | (vx : T)C$$

Example 4 We can write the process

$$P \equiv (va : T)(\mu X.a(x).\overline{x}(x).X | \mu Y.(vb : U)\overline{a}(b).x(y).Y)$$

as

$$C_1[\mu X.a(x).\overline{x}(x).X]$$

where $C_1 = (va : T)[ | \mu Y.(vb : U)\overline{a}(b).x(y).Y)$
### Table 2  Annotated reduction rules

| Rule Type             | Reduction Rule                                                                 |
|-----------------------|-------------------------------------------------------------------------------|
| (COM-ANNOT)           | $\alpha \cdot P(x).P_1 | \overrightarrow{\alpha}(y.q).P_2 \xrightarrow{[\alpha]} P_1[y/x] \mid P_2$ |
| (PAR-ANNOT)           | $P \xrightarrow{\alpha} P'$                                                   |
| (NEW-ANNOT)           | $\overrightarrow{\alpha}(x)P \xrightarrow{\alpha} \overrightarrow{\alpha}(x)P'$ where $T = T'$ if $x \notin n(\alpha)$ $T' = T \downarrow$ if $x \in n(\alpha)$ |
| (UNFOLD-ANNOT)        | $P \alpha \xrightarrow{\alpha} P'$                                          |
| (STRUCT-ANNOT)        | $P \equiv Q \quad Q \xrightarrow{\alpha} Q' \equiv P'' \quad P'' \uparrow P'$ |

### Table 3  The rules for unfolding

| Rule Type   | Reduction Rule                                                                 |
|-------------|-------------------------------------------------------------------------------|
| (UNFOLD)    | $\mu X.P > P[\mu X.P / X]$                                                   |
| (CONTEXT)   | $P > P'$                                                                     |
|             | $\frac{C[P] > C[P']}{P > P'}$                                                 |

or

$$C_2[\mu Y.(v b : U)\overrightarrow{\alpha}(b).x(y).Y] \text{ where } C_2 = (v a : T)(\mu X.a(x).\overrightarrow{\alpha}(b).X \mid [ ])$$.

### 3 Notions of boundedness

We now present the three notions of boundedness introduced by Meyer in [12].

In what follows, we only consider transitions arising from normalized processes, that is, processes without superfluous restrictions. To this end, we define a normalization ordering $\succ$ on processes that removes bound names not found in a process. It is generated by the axiom

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

and closed under structural congruence. A process $P$ is then normalized if it has no superfluous bound names, that is, if $P \nsucc$; we write $P \rightsquigarrow Q$ if $P \succ^* Q$ and $Q$ is normalized. From now on, we assume that in transitions $P \xrightarrow{\alpha} P'$, $P$ is a normalized process.

Note that our notion of normalization is the reason why the structural congruence axiom $(\nu x : T)0 \equiv 0$ often found in presentation of the $\pi$-calculus does not appear here. Its usual justification is that it can be used together with (New-2) to remove superfluous restrictions when applied from left to right; however, when if applied from right to left, such an axiom would introduce spurious restrictions and would also complicate the proof that typability preserves structural congruence.

**Depth-bounded processes** A process $P$ is depth-bounded if every successor process $P'$ reachable from it by zero or more annotated reductions can be rewritten so as to have no more than $k$ nested restrictions. To define this, we first introduce a function $\text{nest}(P)$ 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 [5].

**Definition 4** The nest function is defined by the clauses

\[
\begin{align*}
\text{nest}(0) &= 0 \\
\text{nest}(X) &= 0 \\
\text{nest}((\nu x : T)P) &= 1 + \text{nest}(P) \\
\text{nest}(\mu X.P_1) &= \text{nest}(P_1) \\
\text{nest}(x^p(y).P_1) &= \text{nest}(\overline{x^p(y^q)}.P_1) = 0
\end{align*}
\]

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

**Definition 5** The depth of a process \( P \) is given by

\[
\text{depth}(P) = \min\{\text{nest}(Q) \mid Q \equiv P\}.
\]

**Definition 6** \((\text{Depth-bounded process})\) 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 \( \text{depth}(P') \leq k \).

**Name-boundedness** A process \( P \) is name-bounded if the number of private names occurring in the successors of \( P \) is bounded.

**Definition 7** \((\text{Name-bounded process})\) A process \( P \) is name-bounded if there exists a constant \( k \in \mathbb{N} \) such that whenever \( P \rightarrow^* P' \) and \( P' \leadsto P'' \), then \( P'' \) has at most \( k \) restrictions.

It is obvious that every name-bounded process is also depth-bounded.

**Example 5** The term

\[
P_1 = \mu X.(\nu r_1)(r_1^+(a).X \mid r_1^-(x).X)
\]

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

\[
P_2 = \mu X.(\nu r_1)(\nu r_2)(r_1^+(r_2).X \mid r_1^-(x).X \mid r_2^+(r_1) \mid r_2^-(x))
\]

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

**Width-boundedness** A third notion of boundedness is that of width-boundedness.

**Definition 8** \((\text{Width-bounded process})\) A process \( P \) is width-bounded if there exists a constant \( k \in \mathbb{N} \) such that whenever \( P \rightarrow^* P' \) we have that every bound name in \( P' \) occurs in at most \( k \) parallel components.

Note that this notion coincides with that of fencing recently used by Lange et al. [11] in their analysis of Go programs.

**Boundedness and behavioural equivalences** The notions that we have presented here are orthogonal to usual notions of behavioural equivalence. This can be seen from the following example.

**Example 6** Consider the processes below that all make use of a free name \( y \). We omit polarities and type annotations here.

\[
P_1 = \mu X.(\nu x)(\overline{x^y}\mid x^- (z).X)
\]
Both $P_1$ and $P_2$ are deterministic processes that allow for a single, infinite reduction sequence. However, $P_1$ is name-bounded, whereas every recursive unfolding within $P_2$ introduces two new private names in a new copy of $Q$ and that will never be used in a reduction.

Variations of Example 6 can be used to find examples of equivalent processes that differ wrt. depth-boundedness and width-boundedness.

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 follows that used by Gay and Hole [6] and Bernardi et al. [2].

4.1 Types and type environments

Our type judgements are of the form $\Gamma, \Delta \vdash P$, where $\Gamma$ contains the type bindings of the free polarized names in $P$. A type judgement is to be read as stating that $P$ is well-behaved using the type information found in the type environment $\Gamma$ and the recursion environment $\Delta$ is a set of recursion variables.

Definition 9

A type environment $\Gamma$ is a partial function $\Gamma : N_{pol} \rightarrow T$ such that $\text{dom}(\Gamma)$ is finite. If for every $x \in \text{dom}(\Gamma)$ we have that $\Gamma(x) = T$ with $\text{un} T$, we say that $\Gamma$ is terminal and write $\text{un} \Gamma$.

We are often interested in well-formed judgements. A judgement $\Gamma, \Delta \vdash P$ is well-formed if the type and recursion environments mention exactly the free names and recursion variables occurring in $P$.

Definition 10

(Well-formed judgements) A type judgement $\Gamma, \Delta \vdash P$ is well-formed if for every $n \in \text{dom}(\Gamma)$ either $n \in \text{fn}(P)$ or $\Gamma(n) = T$ with $\text{un} T$ and $\Delta = \text{frec}(P)$.

A type $T = (S_1, S_2)$ is balanced if $S_1 = S_2$. A type environment $\Gamma$ is balanced if for all $x \in \text{dom}(\Gamma)$ we have that either $\Gamma(x) = S$ and $\Gamma(x^-) = S$ or $\Gamma(x) = T$ where $T = B$ or $T = \text{Ch}(T_1)$ for some $T_1$.

We define addition of type environments as in [6].

Definition 11

(Addition of a typed name to a type environment) Let $\Gamma$ be a type environment. Then the addition of a typed name $x$ to $\Gamma$ is defined by

$\Gamma + x^+ : S = \Gamma$, $x^+ : S$ if $x^+ \not\in \text{dom}(\Gamma)$, $x \not\in \text{dom}(\Gamma)$ and $\text{lin} S$

$\Gamma + x^- : S = \Gamma$, $x^- : S$ if $x^- \not\in \text{dom}(\Gamma)$, $x \not\in \text{dom}(\Gamma)$ and $\text{lin} S$

$\Gamma, x : T = \Gamma, x : T$ if $x^+ \not\in \text{dom}(\Gamma)$, $x^- \not\in \text{dom}(\Gamma)$, $x \not\in \text{dom}(\Gamma)$

$(\Gamma + x^e : T) + x^e : T = \Gamma, x : T$ if $\text{un} T$

and is undefined in all other cases. This addition operation extends inductively to a partial binary operation on type environments.
If we know that \( \Gamma = \Gamma_1 + \Gamma_2 \), then Definition 11 tells us if \( x \in \text{dom}(\Gamma) \) then either \( x \in \text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2) \) with polarity \( \varepsilon \) or \( x \) cannot appear with the same polarity in both \( \Gamma_1 \) and \( \Gamma_2 \). If \( \Gamma \) is balanced and \( \Gamma x = \text{lin } T \) for some \( T \), this implies that \( x \) will appear with complementary polarity in \( \Gamma_1 \) and \( \Gamma_2 \).

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

**Definition 12** (Duality of endpoint types) Duality of endpoint types is defined inductively by

\[
!T.S = ?T.\overline{S} \quad \overline{?T.S} = !T.\overline{S} \quad \text{end} = \text{end}
\]

**Definition 13** (Depth of types) The depth of an endpoint type \( S \) is denoted \( d(S) \) and is defined inductively by

\[
d(!T.S) = 1 + d(S) \quad d(?T.S) = 1 + d(S) \quad d(\text{end}) = 0
\]

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

### 4.2 Recursion and recursion environments

In our type system, we must keep track of the recursion variables that are known for a given process expression.

**Definition 14** A recursion environment \( \Delta \) is a set of recursion variables. We let \( \Delta_\emptyset \) denote the empty recursion environment. We let \( \Delta_1 + \Delta_2 \) denote the union \( \Delta_1 \cup \Delta_2 \). We let \( \Delta, X \) denote \( \Delta \cup \{X\} \) if \( X \notin \Delta \).

### 4.3 Type rules

The set of valid type judgements is defined by the rules in Table 4. We now explain the rules.

The rule (In) tells us that an input process will be well-typed if the channel named \( x \) has an input session type \( ?T_1.T_2 \) and the continuation \( P \) is well-typed under the assumption that the type of \( x \) is now \( T_2 \) and the name received has type \( T_1 \). Similarly, the rule (Out) tells us that an output process will be well-typed if the channel named \( x \) has an output session type \( T_1.T_2 \) and the continuation \( P \) is well-typed under the assumption that the type of \( x \) is now \( T_2 \).

Delegation of session names is handled by (Out); session channels are linear: 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) and (Out), the object type \( T_1 \) must be different from \( \text{end} \).

The rules (INCh) and (OUTCh) handle communication on simple channels with type \( \text{Ch}(B) \); here the name carried must be of some base type \( B \).

The rule (Session) constrains the types of private names; they can only be session types. Together with the rules (Rec) and (Var) for recursion this will ensure that channel names are “used up” prior to a recursive unfolding.

The rule (Par) tells us that a parallel composition \( P_1 \parallel P_2 \) is well-typed, if we can partition the type environment and the recursion and use the sub-environments obtained to type \( P_1 \) and \( P_2 \) respectively. In the case of the type environments, this can be used to ensure that the two endpoints of a session channel are placed in separate type environments when we type the parallel components in which they occur.
Table 4 Type rules for depth-boundedness

\[(\text{IN}) \quad \frac{\Gamma, x : T_2, y : T_1, \Delta \vdash P}{\Gamma, x : ?T_1, y : T_1, \Delta \vdash x(y).P} \quad (\text{SESSION}) \quad \frac{\Gamma, \Delta \vdash P}{\Gamma, \Delta \vdash (\nu x : (S, S)) P} \]

where \( T_1 \neq \text{end} \)

\[(\text{OUT}) \quad \frac{\Gamma, x : T_2, \Delta \vdash P}{\Gamma, x : !T_1, T_2, y : T_1, \Delta \vdash x(y).P} \]

where \( T_1 \neq \text{end} \)

\[(\text{InCh}) \quad \frac{\Gamma(x) = \text{Ch}(B)}{\Gamma, y : B, \Delta \vdash P} \quad (\text{OutCh}) \quad \frac{\Gamma(x) = B}{\Gamma, \Delta \vdash \overline{x}(y).P} \]

\[(\text{Par}) \quad \frac{\Gamma_1, \Delta_1 \vdash P_1 \quad \Gamma_2, \Delta_2 \vdash P_2}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2 \vdash P_1 \mid P_2} \]

\[(\text{Nil}) \quad \Gamma, \Delta \vdash 0 \quad \text{where un} \Gamma \]

\[(\text{Var}) \quad \Gamma, \Delta \vdash X \quad \text{if un} \Gamma \text{ and } X \in \Delta \quad (\text{Rec}) \quad \frac{\text{un} \Gamma \quad \Gamma, \Delta, X \vdash P}{\Gamma, \Delta \vdash \mu X. P} \]

The rule (Nil) tells us that all session types must be “used up” if we want to type a terminated process.

The rule (Rec) tells us that a recursive process can only be typed in an unlimited environment; in an unlimited environment, names have the session type end or (end, end).

Finally, the rule (Var) ensures that a recursion variable \( X \) can only be well-typed for \( \Gamma \) and \( \Delta \) if the recursion environment mentions \( X \) and the type environment is unlimited. The latter condition will ensure that the linear names present when a recursion variable \( X \) is reached are no longer available; the existing session channels have been “used up”.

We are often interested in well-formed type derivations, that is, ones that do not mention irrelevant names or recursion variables.

**Definition 15** (Well-formed derivation) A derivation of the judgement \( \Gamma, \Delta \vdash P \) is well-formed if every judgement occurring in the derivation is well-formed.

**Proposition 2** (Well-formedness for depth-boundedness) Consider the type system given in Table 4. Every well-formed judgement \( \Gamma, \Delta \vdash P \) has a derivation if and only if it has a well-formed derivation.

**Proof** By inspection of the type rules; we observe that each rule preserves well-formedness.

In many binary session type systems such as [6] one also allows unlimited channel types of the form \( \text{Ch}(T) \) to appear. The following example illustrates the problem with allowing channels of unlimited type if we want to characterize the property of depth-boundedness.

**Example 7** Consider the process \( P \) that uses an unlimited channel \( s \). Since \( P \) cannot be typed; we leave out type annotations and polarities in its description. Let

\[ P = (\nu n)(\nu v)(\nu a)(\nu x) | \mu S.(s(x).\overline{v}(x)) | (\nu b)((\nu y).\overline{v}(b).\overline{a}(y) | \overline{x}(b)) | S) \]
The process can evolve as follows.

\[ P \rightarrow^* (\nu n)(\nu v)(\nu a)(P_1 | (\nu b)(\nu b')(\nu(\nu(b).\pi(a)) | \nu(b').\pi(b) | \pi(b'))) \]

where \( P_1 = \mu S.(s(x).\bar{v}(x) | (\nu b)((v(y).\bar{v}(b).\pi(y) | \pi(b)) | S)) \) 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 typable process \( P \) will be width-bounded with bound 2, since every name can then occur in either precisely one or precisely two parallel components.

## 5 A subject reduction property

To show our characterization of depth-boundedness, we first establish a type preservation property central to any binary session type system known as fidelity: 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.

### 5.1 Some basic properties of the type system

Since the name used in a reduction may be a private channel, we must also describe how the session types of restricted channels evolve. Every process in which all private names are pairwise distinct gives rise to an internal type environment (Definition 16) 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 16** Let \( P \) be a process whose bound names are pairwise distinct. \( \Gamma_P \) denotes the internal type environment of \( P \); it is defined by the following clauses (where \( \pi \) denotes a prefix).

\[
\begin{align*}
\Gamma_{P_1|P_2} &= \Gamma_{P_1}, \Gamma_{P_2} \\
\Gamma_{\mu X.P} &= \Gamma_P \\
\Gamma_{\nu X: T.P} &= x : T, \Gamma_P \\
\Gamma_X &= \emptyset
\end{align*}
\]

The following lemma tells us that we can discard irrelevant names.

**Lemma 1** (Strengthening) If \( \Gamma, x : S, \Delta \vdash P \) and \( x \notin \text{fn}(P) \), then \( \Gamma, \Delta \vdash P \).

**Proof** A straightforward proof by induction in the type rules. Here are two sample cases; the rest are similar.

**PAR:** Suppose \( \Gamma, x : S, \Delta \vdash P_1 | P_2 \). Then we have wlog the premises \( \Gamma_1, x : S, \Delta_1 \vdash P_1 \) and \( \Gamma_2, x : S, \Delta_2 \vdash P_2 \) where \( \Delta = \Delta_1 + \Delta_2 \) and \( \Gamma = \Gamma_1 + \Gamma_2 + x : S \). By induction hypothesis we have \( \Gamma_1, \Delta_1 \vdash P_1 \) and \( \Gamma_2, \Delta_2 \vdash P_2 \). An application of (PAR) gives us \( \Gamma, \Delta \vdash P_1 | P_2 \).

**IN:** Suppose \( \Gamma, x : S, \Delta \vdash y(z).P \). Then we have that \( x \neq y \) and \( \Gamma, x : S = \Gamma', x : S + y : ?!T_2.T_1 \), so \( \Gamma', x : S + y : T_1 + z : T_2 \vdash P \). By induction hypothesis we have \( \Gamma' + y : T_1 + z : T_2 \vdash P \) and we can apply (IN) to this. \( \square \)
Lemma 2 (Weakening) If un $\Gamma_1$ and $\Gamma, \Delta \vdash P$ with $\text{dom}(\Gamma) \cap \text{dom}(\Gamma_1) = \emptyset$, then also $\Gamma + \Gamma_1, \Delta \vdash P$.

Proof Induction in the type rules. In the case of (VAR) and (NIL), the conclusion is immediate. For (REC), we have that $P = \mu X. P_1$ and by induction hypothesis we have that $\Gamma + \Gamma_1, \Delta, X \vdash P_1$. An application of (REC) to this judgement gives the desired result. For (SESSION), we have by induction hypothesis that $\Gamma + \Gamma_1, x^+: S, x^-: \overline{S} \vdash P$, and we can then use (SESSION) with this judgement as premise. The remaining cases are straightforward. \(\square\)

Lemma 3 (Substitution of variables in typings of recursion) Suppose $\Gamma, \Delta, X \vdash P$ and $\Gamma_1, \Delta \vdash Q$ with $\Gamma_1 \subseteq \Gamma$ and un $\Gamma_1$. Then $\Gamma, \Delta \vdash P\{Q/\chi\}$.

Proof Induction in the structure of $P$.

$P = 0$: Clearly, $0\{Q/\chi\} = 0$, so the result is immediate here.

$P = X$: Here we have $\Gamma, \Delta \vdash P$ with un $\Gamma$. We have $\Gamma_1, \Delta \vdash Q$, but since $\Gamma_1 \subseteq \Gamma$, we have un $\Gamma_1$. We then get from Lemma 2 that $\Gamma, \Delta \vdash Q$.

$P = P_1 | P_2$: We have that

$$\Gamma, \Delta, X \vdash P_1 \mid P_2$$

because

$$\begin{align*}
\Gamma_{11}, \Delta, X &\vdash P_1 \\
\Gamma_{12}, \Delta, X &\vdash P_2
\end{align*}$$

where $\Gamma = \Gamma_{11} + \Gamma_{12}$. By induction hypothesis we have that

$$\begin{align*}
\Gamma_{11}, \Delta, X &\vdash P_1\{Q/\chi\} \\
\Gamma_{12}, \Delta, X &\vdash P_2\{Q/\chi\}
\end{align*}$$

and can now apply the (PAR) rule.

$P = \mu X_2. P_1$: Here we have applied the (REC) rule as

$$\frac{\text{un} \Gamma \quad \Gamma, \Delta, X_2 \vdash P_1}{\Gamma, \Delta \vdash \mu X_2. P_1}$$

and get by induction hypothesis that

$$\Gamma, X_2 \vdash P_1\{Q/\chi\}$$

But we can apply (REC) to this judgement and obtain

$$\Gamma, \Delta \vdash \mu X_2. P_1\{Q/\chi\}$$

$P = (\nu x : (S, \overline{S})) P_1$: We have used the (SESSION) rule

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

and apply the induction hypothesis to the premise of the rule and then (SESSION).
Using session types for reasoning about boundedness...

\[ P = x(y).P_1: \text{ This case is straightforward.} \]

\[ P = \bar{x}(y).P_1: \text{ This case is straightforward.} \]

We also need a substitution lemma for names.

**Lemma 4** (Substitution of names) If \( \Gamma, x : T, \Delta \vdash P \), if the judgement is well-formed and \( y \notin n(P) \cup \text{dom}(P) \) then \( \Gamma, y : T, \Delta \vdash P[y/x] \).

**Proof** A straightforward induction in the type rules.

5.2 A fidelity theorem

To prove fidelity, we must ensure that typability is preserved by unfolding of recursion, by structural congruence and by normalization.

**Lemma 5** (Unfolding) Suppose \( \Gamma \vdash axP \). Then also \( \Gamma \vdash P[\mu x.P / x] \).

**Proof** This follows directly from Lemma 3.

**Lemma 6** (Subject congruence and normalization) Suppose \( \Gamma, \Delta \vdash P \).

1. If \( P \equiv Q \), then also \( \Gamma, \Delta \vdash Q \)
2. If \( P > Q \), then also \( \Gamma, \Delta \vdash Q \)

**Proof** Induction in the rules defining \( \equiv \) and \( > \). All cases are simple for 1. For part 2 on normalization, we must show that if \( \Gamma, \Delta \vdash (\nu x : T)P \) and \( x \notin \text{fn}(P) \), then also \( \Gamma, \Delta \vdash P \). Here we note that the judgement \( \Gamma, \Delta \vdash (\nu x : S)P \) was concluded using (SESSION). We apply the strengthening lemma Lemma 1 to the premise \( \Gamma, \Delta \vdash (\nu x : S)P \) and the result now follows.

The fidelity theorem tells us that if the name \( x \) giving rise to the reduction is free, the annotation of \( x \) in the type environment changes. If \( x \) is private, its annotation in \( (\nu x : T) \) changes to \( (\nu x : T') \), where \( T' = T \downarrow \).

**Theorem 1** (Fidelity) Let \( \Gamma \) be a balanced type environment and let \( P \) be recursion-closed. If \( \Gamma, \Delta \vdash P \) and \( P \xrightarrow{\alpha} P' \) where \( a = n(\alpha) \) then

1. If \( a \in \text{fn}(P) \) and \( \Gamma = \Gamma'', a : T, \) then \( \Gamma', \Delta \vdash P' \) where \( \Gamma' \) is balanced and \( \Gamma' = \Gamma'', a : T \downarrow \)
2. If \( a \notin \text{bn}(P) \cap \text{bn}(P') \), then \( \Gamma, \Delta \vdash P' \) and if \( \Gamma P = \Gamma'', a : T \) then \( \Gamma P' = \Gamma'', a : T \downarrow \) and \( \Gamma P' \) is balanced.

**Proof** Induction in the reduction rules.

(COM-ANNOT) Only the first case is relevant. We know \( P = a^P(x).P_1 \mid \overline{a^P(y^q)}.P_2 \).

Since \( \Gamma, \Delta \vdash P \), we must have that \( \Gamma = \Gamma_1 + \Gamma_2 \) where

\[ \Gamma_1, \Delta \vdash a^P(x).P_1 \quad (1) \]

and

\[ \Gamma_2, \Delta \vdash \overline{a^P(y^q)}.P_2 \quad (2) \]

\( \odot \) Springer
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

$$\Gamma_1', a^p : S, x : T_1, \Delta_\theta \vdash P_1.$$  

Similarly, we must have used (OUT) to conclude (2). Since $\Gamma$ is balanced, we have $\Gamma_2(a^{\overline{p}}) = !T_1.\overline{S}$. By the substitution lemma Lemma 4 and (3), we have $\Gamma_1', a^p : S, \overline{q} : T_1, \Delta_\theta \vdash P_1[\gamma/x]$. Similarly, letting $\Gamma_2 = \Gamma_2', a^{\overline{p}} : !T_1.\overline{S}, \overline{q} : T_1$, we get $\Gamma_2', a^{\overline{p}} : \overline{S}, \Delta_\theta \vdash P_2$. An application of (PAR) now gives us that

$$\Gamma_1' + \Gamma_2' + a^p : S, a^{\overline{p}} : \overline{S}, y : T_1, \Delta_\theta \vdash P_1[\gamma/y] \mid P_2$$

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

(PAR- ANNOT) Since $\Gamma, \Delta_\theta \vdash P \mid Q$, we have that $\Gamma_1, \Delta_\theta \vdash P$ where $\Gamma = \Gamma_1 + \Gamma_2$. The result now follows easily by an application of the induction hypothesis to the reduction $P \overset{a}{\rightarrow} P'$ 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 5 and a direct application of the induction hypothesis.

(STRUCT- ANNOT) Follows from Lemma 6 and a direct application of the induction hypothesis.

\[\square\]

### 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 two 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 7** Let $P$ be an arbitrary recursion-closed process.

1. If $P \overset{[a]}{\rightarrow} P'$, then there exists an unfolding context $C$ and a process $Q$ such that $P \equiv C[Q]$ and $P' \equiv C[Q']$, and $Q \overset{[a]}{\rightarrow} Q'$ is an instance of (COM- ANNOT).
2. If $P \overset{[\text{rec.a}]}{\rightarrow} P'$ then either
   - there exists an unfolding context $C$ and either $P \equiv C[\mu X.Q_1]$ for some $Q_1$ where $Q_1[\mu X.Q_1/X] \overset{[a]}{\rightarrow} Q_1'$ and $P' \equiv C[Q_1']$ or $P \equiv C[(\mu X.Q_1) \mid Q_2]$ where $Q_1[\mu X.Q_1/X] \mid Q_2 \overset{[\text{rec.a}]}{\rightarrow} Q_1' \mid Q_2'$ and $P' \equiv C[Q_1' \mid Q_2']$.
   - or $P \equiv C[\mu X.Q_1] \mid P_1$ where $P \equiv C[Q_2] \mid P_1$ and $\mu X.Q_1 \overset{[a]}{\rightarrow} Q_2$ and $P \overset{[a]}{\rightarrow} P_1$

**Proof** The two cases are proved separately, both by induction in the annotated reduction rules.
First we prove case 1; we present two sample cases. The remaining cases are similar; note that (UNFOLD-ANNOT) cannot apply.

(COM-ANNOT): Here, we have that
\[ a^P(x).P_1 \mid a^P(y^q).P_2 \xrightarrow{[a]} P_1[y^q/x] \mid P_2 \]
but note that \( a^P(x).P_1 \mid a^P(y^q).P_2 \equiv a^P(x).P_1 \mid a^P(y^q).P_2 \mid 0 \), and the result now follows.

(PAR-ANNOT): Here, \( P_1 \mid P_2 \xrightarrow{[a]} P'_1 \mid P_2 \) because \( P_1 \xrightarrow{[a]} P'_1 \). By induction hypothesis, we have that \( P_1 \equiv C[Q] \) and \( Q \xrightarrow{[a]} Q' \) for some \( Q, Q' \). But \( C[Q] \mid P_2 \) is also an unfolding context, and the result now follows.

The proof of Case 2 has a similar structure but uses Case 1 for the case of (UNFOLD-ANNOT).

In what follows, we only consider unfolding contexts that have minimal restriction depth. We can decrease the restriction depth by applying structural congruence axioms. The following observation follows from the formation rules defining unfolding contexts.

**Proposition 3** Let \( C \) be an unfolding context. Every process \( C[\mu X.P] \) is structurally congruent to a process \((\nu x : S)(\mu X.P \mid Q)\) such that \( \text{nest}(Q) \) is minimal.

### 6.2 Nesting properties of well-typed processes

In this subsection we limit our attention to well-typed processes and show two important lemmas concerning the properties of free and bound names in well-typed processes.

The first lemma tells us that names whose session type is terminal disappear.

**Lemma 8** Suppose \( \Gamma, \Delta \vdash P \) where \( \Gamma(c) = (\text{end}, \text{end}) \). Then \( c \notin \text{fn}(P) \).

**Proof** Induction in the structure of \( P \). The only interesting case is that of \( P = \overline{x}(y).P_1 \). Here, the rule (OUT) guarantees that \( x \) cannot be \( c \) but also that \( y \) cannot be \( c \), as the type of a term being output cannot be \( \text{end} \).

The next lemma tells us that names that appear in an unfolding context will not reappear free in the result of unfolding a recursive process. This shows us that session names are “used up” prior to a new recursive call.

**Definition 17** (Known bound names) The set of known bound names in an unfolding context is defined by
\[ \text{kn}([] \mid P) = \emptyset \]
\[ \text{kn}((\nu x : T)C) = \{x\} \cup \text{kn}(C) \]

**Lemma 9** Suppose we have \( \Gamma, \Delta \vdash C[\mu X.P] \). Then we have that \( \text{kn}(C) \cap \text{fn}(\mu X.P) = \emptyset \).

**Proof** If \( \Gamma, \Delta \vdash C[\mu X.P] \), then we also have for some \( \Gamma', \Delta' \) that \( \Gamma', \Delta' \vdash \mu X.P \). Suppose we had a \( c \in \text{fn}(\mu X.P) \) such that also \( c \in \text{kn}(C) \). But \( \Gamma', \Delta' \vdash \mu X.P \) must have been concluded using the type rule (REC), and here we must have that \( \Gamma(c) = (\text{end}, \text{end}) \). But then by Lemma 8 we must have that \( c \notin \text{fn}(\mu X.P) \), which is a contradiction.
6.3 How reductions introduce restrictions

There are two ways in which the nesting depth of restrictions can increase as the result of a
reduction:

- By a communication that reveals new restrictions underneath prefixes.
- By the unfolding of a recursion $\mu X. P$ where $P$ contains restrictions that become revealed.

The type system controls how new restrictions are revealed by ensuring that session types
only allow names to be used a finite number of times (this will take care of the first concern)
and by ensuring that all names have been used up before a recursion is unfolded (this takes
care of the second concern).

We now define two measures that describe the number of restrictions that can be revealed
and then show that the type system limits these.

First we show how to find an upper bound on the number of restrictions that can be revealed
by a recursive unfolding.

**Definition 18** The recursion restriction depth $\text{rrd}(P)$ on processes is defined recursively by the clauses

\[
\text{rrd}(X) = 0 \\
\text{rrd}(0) = 0 \\
\text{rrd}(P_1 | P_2) = \max(\text{rrd}(P_1), \text{rrd}(P_2)) \\
\text{rrd}(x(y).P_1) = \text{rrd}(P_1) \\
\text{rrd}(\pi(y).P_1) = \text{rrd}(P_1) \\
\text{rrd}(\mu X. P_1) = \text{depth}(P_1) \\
\text{rrd}(\nu x . P_1) = \text{rrd}(P_1)
\]

We now show that the recursion restriction depth does not increase because of a recursive
unfolding. First we establish how substitutions of recursion variables can affect the recursion
restriction depth.

**Lemma 10** Let $Q$ be a process. We have that

\[
\text{rrd}(Q\{\mu X. P / x\}) \leq \max(\text{rrd}(Q), \text{rrd}(\mu X. P))
\]

**Proof** The proof proceeds by induction in the structure of $Q$.

- $Q = 0$: We have $\text{rrd}(0) = 0$ and clearly $0 \leq \max(0, \text{rrd}(\mu X. P))$.
- $Q = X$: It is clear that $\text{rrd}(\mu X. P) \leq \max(\text{rrd}(X), \text{rrd}(\mu X. P))$.
- $Q = Q_1 | Q_2$: We have that

\[
Q\{\mu X. P / x\} = Q_1\{\mu X. P / x\} | Q_2\{\mu X. P / x\}
\]

and therefore

\[
\text{rrd}(Q\{\mu X. P / x\}) = \max(\text{rrd}(Q_1\{\mu X. P / x\}), \text{rrd}(Q_2\{\mu X. P / x\})) \\
\leq \max(\max(\text{rrd}(Q_1), \text{rrd}(\mu X. P)), \max(\text{rrd}(Q_2), \text{rrd}(\mu X. P))) \\
= \max(\text{rrd}(Q_1 | Q_2), \text{rrd}(\mu X. P))
\]
Using session types for reasoning about boundedness...

\[ Q = \mu Y.Q_1: \] We have

\[
rrd(Q^{\mu X.P/X}) = rrd(\mu Y.Q_1^{\mu X.P/X}) \\
= \text{depth}(Q_1^{\mu X.P/X}) \\
= \text{depth}(Q_1) \\
= rrd(\mu Y.Q_1) \\
\leq \max(rrd(\mu Y.Q_1), rrd(\mu X.P))
\]

\[ Q = x(y).Q_1: \] Here we have

\[
rrd((x(y).Q_1)^{\mu X.P/X}) = rrd(x(y).Q_1^{\mu X.P/X}) \\
= rrd(Q_1^{\mu X.P/X}) \\
\leq \max(rrd(Q_1), rrd(\mu X.P)) \\
= \max(rrd(x(y).Q_1), rrd(\mu X.P))
\]

\[ Q = \tau(y).Q_1: \] As the previous case.

\[ Q = (\nu x:S)Q_1: \] Here we have

\[
rrd(((\nu x:S)Q_1)^{\mu X.P/X}) = rrd((\nu x:S)Q_1^{\mu X.P/X}) \\
= rrd(Q_1^{\mu X.P/X}) \text{ by induction hypothesis} \\
\leq \max(rrd(Q_1), rrd(\mu X.P)) \\
= \max(rrd((\nu x:S)Q_1), rrd(\mu X.P))
\]

\[ \square \]

**Corollary 1** We have \( rrd(P^{\mu X.P/X}) \leq rrd(\mu X.P) \).

**Proof** Use Lemma 10 with \( P \) in the place of \( Q \). \( \square \)

The recursion restriction depth is easily seen to be invariant under name substitution, since this only involves free names.

**Proposition 4** For any process \( P \) and names \( x \) and \( y \) we have that \( rrd(P) = rrd(P^{\gamma/x}) \).

We have that the recursion restriction depth does not increase as the result of a reduction.

**Lemma 11** If \( P \xrightarrow{\alpha} P' \) then \( rrd(P') \leq rrd(P) \).

**Proof** The proof proceeds by induction in the reduction rules. We present two cases; the remaining cases are similar or simpler.

(COM-ANNOT): Suppose

\[ x(y).P_1 \mid \tau(z).P_2 \xrightarrow{\alpha} P_1^{\gamma/z} \mid P_2 \]
Then clearly
\[
\text{rrd}(x(y).P_1 | \overline{x}(z).P_2) = \max(\text{rrd}(x(y).P_1), \text{rrd}(\overline{x}(z).P_2)) \\
= \max(\text{rrd}(P_1), \text{rrd}(P_2)) \\
= \max(\text{rrd}(P_1[z/y]), \text{rrd}(P_2)) \\
= \text{rrd}(P_1[z/y] | P_2)
\]

(Rec-Annot): The case follows from Corollary 1.

Next, we give an upper bound on the number of restrictions that can be introduced by a reduction step caused by (Com-Annot); since these reductions involve an input and an output prefix, we must measure the maximal number of restrictions that can occur between any two prefixes.

We define the prefix restriction depth \( \text{prd}(P) \) by introducing a subterm relation. We write \( P \overset{k}{\rightarrow} P' \) if \( P' \) is a subterm of \( P \) that can be reached by traversing prefixes passing through \( k \) restrictions (Table 5).

**Example 8** Let
\[
P = \mu X.a(y).(vw)(x^+ \langle y \rangle).\overline{y}^+(z) | y^+(w) | X)
\]

We have
\[
P \overset{0}{\rightarrow} (vw)(x^+ \langle y \rangle).\overline{y}^+(z) | y^+(w) | X) \overset{2}{\rightarrow} X
\]

The set of reachable terms underneath restrictions is then given by the following definition.

**Definition 19** Let \( P \) be a process. We let
\[
\text{reach}(P) = \{ P' | \exists k. P \overset{k}{\rightarrow} P' \}
\]

The prefix restriction depth is the maximal restriction depth of any reachable subterm.

**Definition 20** Let \( P \) be a process. Then
\[
\text{prd}(P) = \max\{ k | \exists P' \in \text{reach}(P). P \overset{k}{\rightarrow} P' \}
\]
Example 9 Let
\[ P = \mu X.(va)(vb)(a^+ b) \mid a^{-y} X \]
Then \( \text{prd}(P) = 2 \).

The prefix restriction depth does not increase along reductions.

Lemma 12 If \( P \xrightarrow{\alpha} P' \) then \( \text{prd}(P') \leq \text{prd}(P) \).

Proof The proof proceeds by induction in the reduction rules. The important case is that of (COM-ANNOT), as this case directly involves prefixes that may reveal restrictions.

Here, we have that
\[ a^p(x).P_1 \mid \overline{a^p}(y^q).P_2 \xrightarrow{\{a\}} P_1[y^q/x] \mid P_2 \]

Note that the subterm relation tells us that \( a^p(x).P_1 \overset{0}{\sim} P_1 \) and that \( \overline{a^p}(y^q).P_2 \overset{0}{\sim} P_2 \). By Definition 20 and the observation that the subterm relation is closed under name substitution, this means that \( \text{prd}(a^p(x).P_1) \geq \text{prd}(P_1) \) and \( \text{prd}(\overline{a^p}(y^q).P_2) \geq \text{prd}(P_2) \). Moreover, we have that \( P_1 \in \text{reach}(a^p(x).P_1 \mid \overline{a^p}(y^q).P_2) \) and \( P_2 \in \text{reach}(a^p(x).P_1 \mid \overline{a^p}(y^q).P_2) \), so by Definition 20 we must have that \( \text{prd}(P_1[y^q/x] \mid P_2) \leq \text{prd}(a^p(x).P_1 \mid \overline{a^p}(y^q).P_2) \). \( \square \)

The following lemma is crucial. It tells us that, after a reduction step, there is an upper bound on the restriction depth of the resulting process.

Lemma 13 Let \( \Gamma \) be balanced. If \( \Gamma, \Delta \vdash P \) and \( P \xrightarrow{\alpha} P' \) then
\[
\text{depth}(P') \leq \text{prd}(P) \cdot \left( \sum_{(T,T') \in \text{ran}(\Gamma')} d(T) + \text{rrd}(P) + \text{depth}(P) \right)
\]

Proof Here we use Lemma 7 and examine how \( \text{depth}(P') \) is related to \( \text{depth}(P) \).

If the reduction was caused by the (COM-ANNOT) axiom and is of the form \( Q \xrightarrow{\delta} Q' \) for the subprocess \( Q \), this can increase the restriction depth by \( \text{prd}(P) \), but by Theorem 1 we see that \( \Gamma', \Delta \vdash P' \) where the depth of the session types of the endpoints of \( x \) decrease by 1 and the depth of all other session types are left unchanged. Moreover we have that \( \text{rrd}(P') \leq \text{rrd}(P) \) and \( \text{rrd}(P') = \text{rrd}(P) \). This means that
\[
\text{depth}(P') \leq \text{prd}(P) + \text{prd}(P) \cdot \sum_{(T,T') \in \text{ran}(\Gamma')} d(T) + \text{rrd}(P) + \text{depth}(P)
\]
\[
\leq \text{prd}(P) \cdot \sum_{(T,T') \in \text{ran}(\Gamma')} d(T) + \text{rrd}(P) + \text{depth}(P)
\]

For the case of recursive unfolding, new restrictions may be introduced by the recursion body. However, by Lemma 9 we see that these names are all distinct from existing bound names, and because of the assumption about the form of unfolding contexts, we know that there exists an unfolding context \( C \) such that
\[ P = (vc : S)(C[\mu X.P_1] \mid Q) \]
for some names \( c \) and some \( Q \) such that
\[ P' = (vc : S)(C[\mu X.P_1] \mid Q') \mid P'[(\mu X.P_1/x)] \]
for some \( Q' \) and \( P' \). That is, the restrictions occurring in the unfolded recursion body can be moved outside. There can be at most \( \text{rrd}(P) \) such restrictions. \( \square \)
Theorem 2 \operatorname{Let} \operatorname{P} \operatorname{be} \operatorname{recursion-closed} \operatorname{and} \operatorname{suppose} \Gamma,\Delta\emptyset \vdash \operatorname{P}. \operatorname{Then} \operatorname{there} \operatorname{exists} \operatorname{a} \operatorname{constant} \operatorname{C} \operatorname{such} \operatorname{that} \operatorname{for} \operatorname{every} \operatorname{P}' \operatorname{where} \operatorname{P} \rightarrow^* \operatorname{P}' \operatorname{we} \operatorname{have} \operatorname{that} \operatorname{depth}(\operatorname{P}') \leq \operatorname{C}.

\textbf{Proof} \quad \operatorname{We} \operatorname{let} \operatorname{C} = \operatorname{prd}(\operatorname{P}) \cdot \sum_{(T,T) \in \operatorname{ran}(\Gamma)} d(T) + \operatorname{rrd}(\operatorname{P}) + \operatorname{depth}(\operatorname{P}). \operatorname{The} \operatorname{result} \operatorname{now} \operatorname{follows} \operatorname{by} \operatorname{a} \operatorname{simple} \operatorname{induction} \operatorname{using} \operatorname{Lemma} \operatorname{13} \operatorname{together} \operatorname{with} \operatorname{Lemma} \operatorname{11} \operatorname{and} \operatorname{Lemma} \operatorname{12}. \quad \square

Corollary 2 \operatorname{Let} \operatorname{P} \operatorname{be} \operatorname{recursion-closed}. \operatorname{Suppose} \Gamma,\Delta\emptyset \vdash \operatorname{P}. \operatorname{Then} \operatorname{P} \operatorname{is} \operatorname{depth-bounded}.

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 when channels are typed with finite session types, 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 private names.

Example 10 The untyped process

\[ \mu X. (\nu r_1)(\nu r_2)(\overline{r_1}(a).X \mid r_1(x).X \mid \overline{r_2}(a) \mid r_2(x)) \]

shows two problems that must be dealt with. Firstly, unfolding a recursion may introduce more parallel recursive components that each have their own private 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 private names that persist—in this case, we get new copies of \((\nu r_2)(\overline{r_2}(a) \mid r_2(x))\) for every unfolding.

The type rules are as in the original type system, with one exception. The intention is that private names can only be introduced into recursive processes. The rule for introducing bound session channels can be seen in Table 6. Note that we also allow the session type \( S \) to be terminated.

Note that this type rule will lead to a type system in which non-recursive processes cannot contain private names; this is uncontroversial, as processes of this kind are always name-bounded.

Furthermore we now modify the notions of addition for type environments and for recursion environments. We add pairs \((\Gamma_1,\Delta_1)\) and \((\Gamma_2,\Delta_2)\) as follows.

\textbf{Definition 21} \quad \operatorname{Let} \Gamma_1, \Gamma_2 \operatorname{be} \operatorname{type} \operatorname{environments} \operatorname{and} \operatorname{let} \Delta_1, \Delta_2 \operatorname{be} \operatorname{recursion} \operatorname{environments}. \quad \operatorname{We} \operatorname{define} \((\Gamma_1, \Delta_1) + (\Gamma_2, \Delta_2) = (\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2)\) \operatorname{if} \Delta_1 = \Delta_0 \operatorname{or} \Delta_2 = \Delta_0.

This modified definition of addition will ensure that when we type a parallel composition \( P_1 \mid P_2 \), all recursion variables will appear in \( P_1 \) or \( P_2 \). This will in particular prevent nested recursions such as \( \mu X. (\nu x)(\overline{x}(w).X \mid \mu Y.x(z).Y) \) that allow the creation of an unbounded number of bound names using \((\nu x)\).

The revised type system still satisfies the well-formedness property.
Proposition 5 (Well-formedness for name-boundedness) Consider the type system for name-boundedness. Every well-formed judgement $\Gamma, \Delta \vdash P$ has a derivation if and only if it has a well-formed derivation.

Proof By inspection of the type rules; we observe that each rule preserves well-formedness. $\square$

7.1 Fidelity

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

Theorem 3 (Fidelity) Let $\Gamma$ be a type environment. If $\Gamma, \Delta, \emptyset \vdash P$ and $P \xrightarrow{\cdot} P'$ then

- if $x \in \text{fn}(P)$ and $\Gamma = \Gamma'', x : T$, then $\Gamma', \Delta_{\emptyset} \vdash P'$ where $\Gamma'$ is balanced and $\Gamma'' = \Gamma'', x : T \downarrow$
- if $x \notin \text{fn}(P)$, 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 $P$ is well-typed in a limited environment, then it is name-bounded. More precisely, we 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 private names

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 14 Let $\mu X.P$ be a process for which all binding occurrences of recursion variables are distinct. If $\Gamma, \Delta \vdash \mu X.(\nu n)(C_1[X] \mid C_2[X] \mid P')$, 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 $\mu X.P = \mu X.(\nu n)(C_1[X] \mid C_2[X] \mid P')$ where $n$ is a set of names (possibly empty), and $C_1$ and $C_2$ are process contexts.

The derivation of the type judgement $\Gamma, \Delta \vdash \mu X.(\nu n)(C_1[X] \mid C_2[X] \mid P')$ must have used the (REC) type rule in its final step, having premise $\Gamma, \Delta, X : \Gamma \vdash (\nu n)(C_1[X] \mid C_2[X] \mid P')$. 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 $\Gamma_2$ is unlimited. However, there can be no well-formed derivation of the latter, since an eventual application of (VAR) would require that $X \notin \Delta_{\emptyset}$.

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

This lemma tells us that there can be no finite, non-recursive subprocesses of a recursive process that have names bound by a restriction that are only found there; any such bound name found in a non-recursive subprocess will also appear in the recursive part of the process.
Lemma 15 Let \( P = \mu X.(C[X] \mid Q) \). If \( \Gamma, \Delta \vdash P \) is well-formed, if \( C[X] \) is an unfolding context, then for every \( n \in \text{pn}(P) \cap \text{n}(Q) \) whose type is not a terminated type, we also have that \( n \in \text{n}(C[X]) \).

**Proof** Since \( \Gamma, \Delta \vdash P \), we must have that \( \Gamma_Q, \Delta_Q \vdash Q \) for some \( \Gamma_Q, \Delta_Q \). Consider a name \( n \in \text{pn}(P) \cap \text{n}(Q) \). Suppose \( n \notin \text{n}(C[X]) \). We would then have a subprocess \((\nu n : T)P'\) of \( Q \) that would contain both \( n^+ \) and \( n^- \). This subprocess would be typed using (SESSION) and \( \Gamma_Q', n^+ : S, n^- : S', \Delta_{P'} \vdash P' \) for some \( S, S' \) and \( \Gamma_Q', \Delta_{P'} \). However, since the type derivation is well formed, some recursion variable would then have to appear in \( \Delta_{P'} \) and therefore also in the recursion environment \( \Delta_Q \) in the judgement \( \Gamma_Q, \Delta_Q \vdash Q \). But the type of \( n \) is not terminated, so\((\text{PAR})\) tells us that this cannot be the case. By contradiction, we must have that \( n \in \text{n}(C[X]) \). \(\square\)

If \( P \) is a process, we say that a subprocess \( \mu X.P_1 \) is a **recursion instance** in \( P \). We let \( \text{recs}(P) \) denote the number of recursion instances in \( P \).

**Example 11** In the process (we here omit the polarities of names)

\[
P = (\mu X.x(y), \overline{z}(X \mid X)) \mid (\mu Y.(vz)\overline{z}(z(w)).Y)
\]

we have the recursion instances \( \{\mu X.x(y), \overline{z}(z), \mu Y.(vz)\overline{z}(z).z(w)\} \), so \( \text{recs}(P) = 2 \).

We now show that the number of recursion instances that will appear in any reduction sequence for a well-typed process is bounded.

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

**Proof** Induction in the reduction rules. \(\square\)

**Lemma 17** Suppose \( \Gamma, \Delta \vdash P \) and \( P > P_1 \). Then \( \text{recs}(P) \geq \text{recs}(P_1) \).

**Proof** For a recursion unfolding \( \mu X.P > P[\mu X.P / X] \), the definition of \( \text{recs}(P) \) tells us that the number of recursion instances can only increase if \( P[\mu X.P / X] \) contains more parallel components than \( \mu X.P \). Consequently we must have \( P_1 = \mu X.P \mid \mu X.P \mid P' \) for some \( P' \), that may contain further recursion instances. However, we know that both of these instances of \( \mu X.P \) arise from unfoldings of \( X \).

When typing \( \mu X.P \), we must have used the type rule (\( \text{PAR} \)) with the premise \( \Gamma, \Delta \vdash X \mid X \mid P' \). By the (\( \text{PAR} \)) rule, wlog that we would then have to type this with premises \( \Gamma_1, \Delta \vdash X \) and \( \Gamma_2, \Delta_{\theta} \vdash X \mid P' \) with \( \Gamma = \Gamma_1 + \Gamma_2 \), but the type judgement \( \Gamma, \Delta_{\theta} \vdash X \mid P' \) cannot be typed, as this would have involved another use of (\( \text{PAR} \)) with \( \Gamma_{11}, \Delta_{\theta} \vdash X \) for some \( \Gamma_{11} \) and where \( X \notin \Delta_{\theta} \).

In conclusion, we must have that the number of recursion instances cannot increase when unfolding a well-typed process. However, Lemma 14 tells us that there can be at most one occurrence of the recursion variable in a well-typed recursive process. \(\square\)

The following normal form theorem uses the lemmas just established to state that there is a uniform bound on the number of recursion instances in any reduction sequence starting in a well-typed process \( P \).
Theorem 4 If $\Gamma, \Delta \vdash P$, then there exists a $k \geq 0$ such that whenever $P \rightarrow^* P'$, we have $P' \equiv P_1 \mid P_2$ where $\text{recs}(P_1) \leq k$, $\text{recs}(P_2) = 0$ and $P_2$ contains no restrictions.

Proof We show that for all $n \geq 1$, if $P \rightarrow^n P'$, then we have $P' \equiv P_1 \mid P_2$ where $\text{recs}(P_1) \leq k$, $\text{recs}(P_2) = 0$ and $P_2$ contains no restrictions. We have that $k = \text{recs}(P)$ and the proof now proceeds by induction in $n$.

$n = 1$: Here we proceed by induction in the type derivation of $\Gamma, \Delta \vdash P$. We consider each rule in turn.

(IN) and (OUT): Neither 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, using Lemmas 16 and Lemma 17.

\[ \square \]

Theorem 5 If $\Gamma, \Delta \vdash P$, then $P$ is name-bounded.

Proof By Theorem 4, there is a $k \geq 0$ such that if $\Gamma, \Delta \vdash P$, whenever $P \rightarrow^* P'$, 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 $\text{depth}(P') \leq d$ for any such $P'$.

By Lemma 15, we have that 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 $P'$, where $P'' = (\nu x_1)\ldots(\nu x_d)P''$ for some $P'''$ that does not contain restrictions at the outermost level and such that $\{x_1, \ldots, x_d\} \subseteq \text{fn}(P''')$. Moreover, for some $k' \leq k$ we have $P''' = P_1'' \mid \cdots \mid P_k'' \mid P_{k'+1}''$ where $P_1'', \ldots, P_k''$ contain recursion instances and, because of Theorem 4, $P_{k'+1}''$ is a process not containing recursion instances. We know that for some $d$ there are at most $d \cdot k$ bound names in $P'$.

\[ \square \]

8 The relation to other classes of processes

Because of the use of binary session types, every typable process in our systems will be width-bounded with 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. As an example, consider the process $P_1$ with free names $b$, $c$ and $d$ given by

\[ P_1 \overset{\text{def}}{=} (\mu X.(va:S)(\overline{a}(b) \mid a(x).(X \mid \overline{a}(c).b(x) \mid \overline{b}(c).d(x)))) \]

where $S = (\!\!T_B.\!\!\text{end}, \!\!?T_B.\!\!\text{end})$ and assume the type environment $\Gamma$ where $\Gamma(c) = B$ for some type $B$ and $\Gamma(b) = \Gamma(d) = T_B$, with $T_B = (\!\!B.\!\!\text{end}, \!\!?B.\!\!\text{end})$. We then have that $\Gamma, \Delta_\emptyset \vdash P_1$.

This is not a finite-control process [4], since the reduction sequence

\[ P \rightarrow^k P_1 \mid \overline{a}(c).b(x) \mid \overline{b}(c).d(x) \mid \cdots \mid \overline{a}(c).b(x) \mid \overline{b}(c).d(x) \]

that results in $k-1$ parallel components, shows that the number of parallel components along a computation can be unbounded for a well-typed process. This also means that $P_1$ is not a
bounded process in the sense of [3]. On the other hand, $P_1$ is depth-bounded. 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 $\pi$-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.

The paper by D’Osualdo and Ong [5] provides a different type system for characterizing depth-bounded processes. The crux of their type system, which considers a $\pi$-calculus with guarded replication instead of explicit recursion, is to characterize name usage by establishing properties of the forest structure of $\pi$-calculus terms. As in our approach, normal forms with the structure $((\nu n)(P_1 | \cdots | P_n))$ are important, but the focus is now on characterizing how and when names can be shared by and extruded from a parallel component in this normal form. In our type system, we instead use the behavioural properties of binary session types to control when and how many times a name can be used.

Another difference is that in [5] a type inference algorithm is proposed that provides a safe bound on the restriction depth for depth-bounded processes, while our type system characterizes this bound by Lemma 13.

A precise study of the relationship between our type system and that of D’Osualdo and Ong is a topic for further work.

A further topic of investigation is to adapt the type inference algorithm proposed in [7] 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 $\pi$-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 relationship between our type system for depth-boundedness and the type system due to D’Osualdo and Ong [5], more precisely the precise relationship between the classes of well-typed processes in the two systems.

References

1. Amadio, R.M., Meyssonnier, C.: On decidability of the control reachability problem in the asynchronous pi-calculus. Nordic J. Comput. 9(2), 70–101 (2002)
2. Bernardi, G., Dardha, O., Gay, S.J., Kouzapas, D.: On duality relations for session types. In: Maffei, M., Tuosto, E. (eds.) Trustworthy Global Computing, pp. 51–66. Springer, Berlin (2014)
3. Caires, L: Behavioral and spatial observations in a logic for the pi-calculus. In: Walukiewicz, I (ed.) Proceedings of FOSSACS 2004, LNCS 2987, pp. 72–89, Springer, Berlin. https://doi.org/10.1007/978-3-540-24727-2_7
4. Dam, M.: Model checking mobile processes. Inf. Comput. 129(1), 35–51 (1996). https://doi.org/10.1006/inco.1996.0072
5. D’Osualdo, E., Ong, L.: A type system for proving depth boundedness in the pi-calculus. CoRR (2015). arXiv:1502.00944
6. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. Acta Inf. 42(2–3), 191–225 (2005). https://doi.org/10.1007/s00236-005-0177-z
7. Graversen, E.F., Harbo, J.B., Hüttel, H., Bjerregaard, M.O., Poulsen, N.S., Wahl, S.: Type inference for session types in the $\pi$-calculus. In: Hildebrandt, T., Ravara, A., van der Werf, J., Weidlich, M. (eds.) Proceedings of WS-PM 2014 and WS-FM/BEAT 2015, LNCS 9421, Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-33612-1_7
8. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Proceedings of ESOP’98, pp. 122–138 (1998). https://doi.org/10.1007/BFb0053567
9. Hüchting, R., Majumdar, R., Meyer, R.: A Theory of Name Boundedness, pp. 182–196. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-40184-8_14
10. Lanese, I., Pérez, J.A., Sangiorgi, D., Schmitt, A.: On the expressiveness and decidability of higher-order process calculi. Inf. Comput. 209(2), 198–226 (2011). https://doi.org/10.1016/j.ic.2010.10.001
11. Lange, J., Ng, N., Toninho, B., Yoshida, N.: Fencing off Go: Liveness and safety for channel-based programming. In: POPL 2017, pp. 748–761. ACM, New York (2017). https://doi.org/10.1145/3009837
12. Meyer, R.: On boundedness in depth in the pi-calculus. In: Ausiello, G., Karhumäki, J., Mauri, G., Luke Ong, C.-H. (eds.) Proceedings of TCS 2008, Volume 273 of IFIP, pp. 477–489. Springer, Berlin (2008). https://doi.org/10.1007/978-0-387-09680-3_32
13. Meyer, R., Gorrieri, R.: On the relationship between $\pi$-calculus and finite place/transition Petri nets. In: Bravetti, M., Zavattaro, G. (eds.) Proceedings of CONCUR 2009. LNCS, Vol. 5710, pp. 463–480. Springer, Heidelberg (2009)
14. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I. Inf. Comput. 100(1), 1–40 (1992). https://doi.org/10.1016/0890-5401(92)90008-4
15. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, II. Inf. Comput. 100(2), 41–77 (1992). https://doi.org/10.1016/0890-5401(92)90009-5
16. Owicki, S., Lamport, L.: Proving liveness properties of concurrent programs. ACM Trans. Program. Lang. Syst. 4(3), 455–495 (1982). https://doi.org/10.1145/357172.357178
17. Wies, T., Zufferey, D., Henzinger, T.A.: Forward analysis of depth-bounded processes. In: Luke Ong (ed.) Proceedings of the 13th International Conference on Foundations of Software Science and Computational Structures (FOSSACS’10), pp. 94–108. Springer, Berlin. https://doi.org/10.1007/978-3-642-12032-9_8

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.