Mobile Processes with Dependent Communication Types and Singleton Types for Names and Capabilities*

Torben Amtoft[1,**] and J. B. Wells[2]

[1] Kansas State University
[2] Heriot-Watt University

Abstract. There are many calculi for reasoning about concurrent communicating processes which have locations and are mobile. Examples include the original Ambient Calculus and its many variants, the Seal Calculus, the MR-calculus, the M-calculus, etc. It is desirable to use such calculi to describe the behavior of mobile agents. It seems reasonable that mobile agents should be able to follow non-predetermined paths and to carry non-predetermined types of data from location to location, collecting and delivering this data using communication primitives. Previous type systems for ambient calculi make this difficult or impossible to express, because these systems (if they handle communication at all) have always globally mapped each ambient name to a type governing the type of values that can be communicated locally or with adjacent locations, and this type can not depend on where the ambient has traveled.

We present a new type system PolyA where there are no global assignments of types to ambient names. Instead, the type of an ambient process P not only indicates what can be locally communicated but also gives an upper bound on the possible ambient nesting shapes of any process P' to which P can evolve, as well as the possible capabilities and names that can be exhibited or communicated at each location. Because these shapes can depend on which capabilities and names are actually communicated, the types support this with explicit dependencies on communication. PolyA is thus the first type system for an ambient calculus which provides type polymorphism of the kind that is usually present in polymorphic type systems for the λ-calculus.

1 Introduction

1.1 Background and Motivation. Recently, many calculi have been proposed for processes and mobility: while the π-calculus [15] is probably still the most popular, the ambient calculus [7] has gained quite some popularity, with

* Partially supported by EC FP5 grant IST-2001-33477, EPSRC grant GR/L 36963, and Sun Microsystems equipment grant EDUD-7826-990410-US.

** Most of the work was done while Amtoft was at Heriot-Watt University paid by EC FP5 grant IST-2001-33477.

several other frameworks joining the competition. These calculi typically send *names* in messages, so some kind of dependent typing seems needed to get a precise analysis. This paper explores this idea.

To have a concrete model, we use a variant ambient calculus containing the features of the original Mobile Ambients (MA) [7] and Boxed Ambients (BA) [4], in the latter of which messages can be sent not only between processes in the same ambient, but also between parent and child ambients. It should be easy to extend our new PolyA type system to the Seal calculus [19], *mutatis mutandis*. The issues addressed by the Join calculus [11] are largely orthogonal to those that we raise.

The Ambient Calculus, a process calculus designed by Cardelli and Gordon [7], and later extended and modified by many others, models these notions:

Location: Processes are located in *ambients* which can be nested in a tree.
Mobility: Ambients can move, making the tree dynamic.
Communication: Processes that are "close" to each other can exchange values.

As an example, consider this process:

$$q_1[p_1[(x)^\star.x[\mathsf{out}\ p_1.\mathbf{0}]] \mid \langle \mathsf{in}\ p_1, r\rangle^\star.\mathbf{0}] \mid q[\mathsf{in}\ q_1.\mathbf{0} \mid (p,v)^\uparrow.p.\langle v\rangle^\uparrow.\mathbf{0}]$$

The example ambient named q is the most simple kind of *generic mobile agent*, namely a *messenger*. That is, q first goes somewhere looking for messages to deliver, then q collects a destination and a payload, and then q goes to that destination and delivers that payload. This is perhaps the simplest example of a generic mobile agent. This rewriting sequence shows the behavior of q:

$$\begin{aligned}
& q_1[p_1[(x)^\star.x[\mathsf{out}\ p_1.\mathbf{0}]] \mid \langle \mathsf{in}\ p_1, r\rangle^\star.\mathbf{0}] \mid q[\mathsf{in}\ q_1.\mathbf{0} \mid (p,v)^\uparrow.p.\langle v\rangle^\uparrow.\mathbf{0}] \\
\longrightarrow\ & q_1[p_1[(x)^\star.x[\mathsf{out}\ p_1.\mathbf{0}]] \mid \langle \mathsf{in}\ p_1, r\rangle^\star.\mathbf{0} \mid q[(p,v)^\uparrow.p.\langle v\rangle^\uparrow.\mathbf{0}]] \\
\longrightarrow\ & q_1[p_1[(x)^\star.x[\mathsf{out}\ p_1.\mathbf{0}]] \mid q[\mathsf{in}\ p_1.\langle r\rangle^\uparrow.\mathbf{0}]] \\
\longrightarrow\ & q_1[p_1[(x)^\star.x[\mathsf{out}\ p_1.\mathbf{0}] \mid q[\langle r\rangle^\uparrow.\mathbf{0}]]] \\
\longrightarrow\ & q_1[p_1[r[\mathsf{out}\ p_1.\mathbf{0}] \mid q[\mathbf{0}]]] \\
\longrightarrow\ & q_1[p_1[q[\mathbf{0}]] \mid r[\mathbf{0}]]
\end{aligned}$$

In the first step, q moves into q_1; in the second step, q receives from its parent q_1 the path $\mathsf{in}\ p_1$ and the payload r; in the third step, q follows its assigned path and moves into p_1; in the fourth step, q delivers its payload r to p_1 which constructs an ambient with the name r; in the fifth and final step, this ambient moves out of p_1.

Nearly all type systems for ambient calculi follow the example of the seminal system of Cardelli and Gordon [8] and assign each ambient name a "topic of conversation": if the name a is assigned the type $\mathsf{Amb}[T]$ then within ambients named a only values of type T can be communicated. For BA, this must be generalized as done in [4]: if the name a is assigned the type $\mathsf{Amb}[T_1, T_2]$ then ambients named a have the property that their "internal" topic of conversation is T_1 and their "upwards" topic of conversation is T_2. Still, in order to type

our example program even that approach has to be extended; observe that the process inside q has two upwards topic of communication: one with arity one, and another with arity two. Clearly, they cannot be mixed up, so it is "safe" to introduce union types with one component for each possible arity. With this extension, our example can be typed like this:

$$\begin{aligned}
\text{type of } r, v, x = T_0 &= \mathsf{Amb}[\mathsf{shh}, \mathsf{shh}] \\
\text{type of } p_1 &= \mathsf{Amb}[T_0, \mathsf{shh}] \\
\text{type of } p &= \mathsf{Cap}[T_0] \\
T_1 &= T_0 \cup (\mathsf{Cap}[T_0] \times T_0) \\
\text{type of } q &= \mathsf{Amb}[\mathsf{shh}, T_1] \\
\text{type of } q_1 &= \mathsf{Amb}[T_1, \mathsf{shh}]
\end{aligned}$$

Unfortunately, the previous type systems are quite inflexible about allowing generic functionality. Consider the previous example process extended to have *two* possible execution paths, in that q can enter either q_1 or q_2:

$$\begin{aligned}
& q_1[\langle \mathsf{in}\ p_1, r\rangle^\star.\mathbf{0} \mid p_1[(x)^\star.x[\mathsf{out}\ p_1.\mathbf{0}]]] \ (x \textit{ must be a name}) \\
\mid\ & q_2[\langle \mathsf{in}\ p_2, \mathsf{out}\ p_2\rangle^\star.\mathbf{0} \mid p_2[(x)^\star.r[x.\mathbf{0}]]] \ (x \textit{ must be a capability}) \\
\mid\ & q[\mathsf{in}\ q_1.\mathbf{0} \mid \mathsf{in}\ q_2.\mathbf{0} \mid (p, v)^\uparrow.p.\langle v\rangle^\uparrow.\mathbf{0}]
\end{aligned} \tag{1}$$

Here, the messenger q must be able to deliver two different types of payloads, *both* an ambient name *and* a capability. None of the previous type systems for ambient calculi allow this. In general, the previous type systems do not support the possibility that a mobile agent may carry non-predetermined types of data from location to location and deliver this data using communication primitives. Polymorphic type systems for the λ-calculus have no trouble with this kind of generic functionality. In previous type systems for ambient calculi, generic mobile agents can be encoded by using extra ambient wrappers, one for each type of data to be delivered. Each location would be careful to only look inside arriving ambients with the correct name. However, this loses the ability to predict whether the correct type of data is being delivered to each location. In solving this problem, a key observation is that the topic of conversation within q depends on whether q is inside q_1 or inside q_2, so some form of *dependent* type seems to be needed.

1.2 Our New Type System. To overcome the weaknesses of previous type systems for generic functionality, we will present a new type system PolyA. Types will indicate the possible positions of capabilities, inputs, and outputs, and also represent upper bounds on the possible ambient nesting tree into which a process can evolve. Thus they look much like processes, as is also the case for, e.g., the types of [9]. Here is an example judgement derivable in PolyA:

$$a[\mathsf{in}\ b.\mathbf{0}] \mid b[\mathbf{0}] \quad : \quad a[\mathsf{in}\ b] \mid b[a[\mathsf{in}\ b]]$$

The types say nothing about the number of copies of a feature at a location, unlike what is the case in [18]. Features of our systems are as follows:

– There are singleton types of ambient names and explicit dependencies on communication, as illustrated by this judgement:

$$(x)^\star.x[\mathbf{0}] \mid \langle a\rangle^\star.\mathbf{0} \quad : \quad ((x)^\star \to x[\mathbf{0}]) \mid \langle a\rangle^\star \mid a[\mathbf{0}]$$

– Sequential composition is approximated in the types by parallel composition, except for inputs, e.g.:

$$p[\mathsf{in}\ q.\mathsf{in}\ r.\mathbf{0}] \mid r[\mathbf{0}] \quad : \quad p[\mathsf{in}\ q \mid \mathsf{in}\ r] \mid r[p[\mathsf{in}\ q \mid \mathsf{in}\ r]]$$

We deliberately collapse most sequencing information to keep the analysis tractable. Also, Amtoft (one of this paper's authors), Kfoury, and Pericas [3, 2] have already investigated increasing precision by keeping track of the sequence of actions, and we do not have a new contribution to make on that topic.

– The types merge distinct ambients at a location with the same name:

$$a[T_1] \mid a[T_2] \doteq a[T_1 \mid T_2]$$

Merging does not occur for same-arity inputs or outputs.

– Types can be infinitely deep trees, e.g.:

$$!a[!\mathsf{in}\ a.\mathbf{0}] \quad : \quad \mathsf{letrec}\ X = a[\mathsf{in}\ a \mid X]\ \mathsf{in}\ X$$

We only consider types that can be given a finite term representation. Due to binders, their precise characterization is non-trivial [12].

– For our convenience, basically we employ only one sort of types which is used for both processes and messages (a.k.a. expressions) since a message M can be viewed as a process $M.\mathbf{0}$.

– Unlike previous type systems, there are no type assumptions for the names of ambients. Instead, information on the topics of conversation inside various ambients is put in the types of processes.

– Also unlike previous type systems, there are no type assumptions for the types of the bound variables of input processes. If you like, for comparison with previous type systems, you can assume that there is a type assumption $x : \alpha_x$ for each such variable x and that each occurrence of x in the types is replaced by α_x.

PolyA can assign the example process in (1) the following type:

$$\begin{array}{rl}
\mathsf{letrec}\ X_0 = & \mathsf{in}\ \{q_1, q_2\} \mid (p, v)^\uparrow \to (p \mid \langle v\rangle^\uparrow) \\
X_1 = & q[X_0 \mid \mathsf{in}\ p_1 \mid \langle r\rangle^\uparrow] \mid r[\mathsf{out}\ p_1] \\
X_2 = & q[X_0 \mid \mathsf{in}\ p_2 \mid \langle \mathsf{out}\ p_2\rangle^\uparrow] \mid r[\mathsf{out}\ p_2] \\
X = & \ \ q_1[X_1 \mid \langle \mathsf{in}\ p_1, r\rangle^\star \mid p_1[X_1 \mid (x)^\star \to x[\mathsf{out}\ p_1]]] \\
& \mid q_2[X_2 \mid \langle \mathsf{in}\ p_2, \mathsf{out}\ p_2\rangle^\star \mid p_2[X_2 \mid (x)^\star \to r[x]]] \\
& \mid q[X_0] \\
\mathsf{in}\ \ X &
\end{array}$$

This proves that the example process has only well defined behavior, something which no previous type system for ambients can do.

Names	a, b	$\in$ Names	(a countably infinite set)
Directions	λ	$\in$ Dirs	$::= \star \mid \uparrow \mid \downarrow a$
Constants	c	$\in$ Const	$::= 0 \mid 1 \mid 2 \mid \ldots \mid \mathsf{true} \mid \mathsf{false}$
Expressions	M	$\in$ Exp	$::= a \mid c \mid \mathsf{in}\ a \mid \mathsf{out}\ a \mid \mathsf{open}\ a \mid \epsilon \mid M_1.M_2$
Processes	P, Q, R	$\in$ Proc	$::= \mathbf{0} \mid P_1 \mid P_2 \mid !P$
			$\mid (\nu a).P \mid M.P \mid a[P]$
			$\mid (\vec{a})^{\lambda}.P \mid \langle \vec{M} \rangle^{\lambda}.P$

Fig. 1. The Soft-boxed Ambient Calculus, process syntax.

2 The Soft-Boxed Ambient Calculus

Our soft-boxed ambient calculus is given in Fig. 1. It has the features of BA [4] and the original MA [7], because this allows us to discuss issues of typing both with and without the open capability. Already [6] observed that the presence of open complicates analysis significantly (thus motivating [4]). Accordingly, in Sect. 4.2 we shall see that an anomaly in the types when open is used makes the types look a bit ugly, but does not seem likely to cause difficulty in practice in showing the well-definedness of safe processes.

Our calculus does not include "co-capabilities". These originated in [14] and have since then become quite popular (e.g., they were added to BA in [5]), partly because their presence has often significantly improved analysis precision and partly because they make it easier to code the desired behavior and avoid *grave interferences*. We do not consider co-capabilities because they do not pose any difficulties for our methods and because they seem unlikely to improve analysis precision, for the following reasons. Co-capabilities may be used to say "we only allow someone in *after* something has happened", but in this paper PolyA deliberately collapses information about sequencing other than input actions. PolyA could be changed not to collapse this information, but the additional precision gained is orthogonal to the main points we make about type polymorphism. Co-capabilities may also be used to say "this ambient named a allows someone to enter whereas that ambient also named a does not allow anyone to enter" but PolyA already keeps separate information about multiple ambients with the same name at different locations and deliberately collapses the information about multiple ambients with the same name which are also at the same location.

As in [4], we have synchronous output $\langle \vec{M} \rangle^{\lambda}.P$; asynchronous output (as in the original ambient calculus) is a special case with $P = \mathbf{0}$. We allow constants, as they are useful for writing examples. Extending PolyA to allow operations on constants is straightforward. Note that $c.P$, though intuitively ill-formed, is in fact a member of Proc. PolyA could be extended to ensure that well typed processes never evolve into such configurations, but for simplicity we have refrained from doing so.

$\mathsf{fn}(a)$	$= \{a\}$	$\mathsf{fan}(a)$	$= \emptyset$
For $f \in \{\mathsf{fn}, \mathsf{fan}\}$:			
$f(\star)$	$= \emptyset$		
$f(\uparrow)$	$= \emptyset$	$f(\downarrow a)$	$= \{a\}$
$f(c)$	$= \emptyset$	$f(\mathsf{in}\ a)$	$= \{a\}$
$f(\mathsf{out}\ a)$	$= \{a\}$	$f(\mathsf{open}\ a)$	$= \{a\}$
$f(\epsilon)$	$= \emptyset$	$f(M_1.M_2)$	$= f(M_1) \cup f(M_2)$
$f(\mathbf{0})$	$= \emptyset$	$f(P_1 \mid P_2)$	$= f(P_1) \cup f(P_2)$
$f(!P)$	$= f(P)$	$f((\nu a).P)$	$= f(P) \setminus \{a\}$
$f(M.P)$	$= f(M) \cup f(P)$	$f((\vec{a})^{\lambda}.P)$	$= (f(P) \setminus \{\vec{a}\}) \cup f(\lambda)$
$f(a[P])$	$= \{a\} \cup f(P)$	$f(\langle \vec{M} \rangle^{\lambda}.P)$	$= f(M) \cup f(\lambda) \cup f(P)$

Fig. 2. Free names and free ambient names for source terms.

We often write M for $M.\mathbf{0}$, $\langle \vec{M} \rangle^{\lambda}$ for $\langle \vec{M} \rangle^{\lambda}.\mathbf{0}$, $(\vec{a}).P$ for $(\vec{a})^{\star}.P$, and $\langle \vec{M} \rangle.P$ for $\langle \vec{M} \rangle^{\star}.P$. We identify processes that are equal modulo consistent renaming of bound variables.

We employ functions $\mathsf{fn}(\lambda)$, $\mathsf{fn}(M)$, and $\mathsf{fn}(P)$, finding the free names; and functions $\mathsf{fan}(\lambda)$, $\mathsf{fan}(M)$, and $\mathsf{fan}(P)$, finding the free names that occur in "ambient naming position". A member of $\mathsf{fan}(P)$ can only be replaced by another name, not by an arbitrary expression. The only difference between $\mathsf{fan}()$ and $\mathsf{fn}()$ is in the case for $M = a$; therefore we can present these functions in a compact way as done in Fig 2.

We write $X[a := M]$ for the result of substituting M for a in X, where X is either a process P, an expression M, or a direction λ. Let the notation $X[\vec{a} := \vec{M}]$ stand for simultaneous substitution of $M_1, \ldots, M_n$ respectively for $a_1, \ldots, a_n$ in X. Substitution is performed using the usual renaming of bound variables. For example, if $P = (\nu a).(a[\mathsf{in}\ b])$ then $P[b := a] = (\nu a').(a'[\mathsf{in}\ a])$ where a' is "fresh".

A key observation is that $P[a := M]$ is not always defined because the result may not qualify as a member of Proc. For example, $a[Q][a := \mathsf{in}\ b]$, which would seem to result in the ill-formed $(\mathsf{in}\ b)[Q]$, is actually undefined. A major task of PolyA is to ensure that all substitutions performed at runtime are well-defined, cf. rule (Red Comm) in Fig 3. On the semantic level, the function fan provides us with a necessary and sufficient condition that a process does not "go wrong".

Lemma 2.1. *$P[a := M]$ is well-defined if and only if $a \in \mathsf{fan}(P)$ implies that $M = b$ for some name b.* □

The rewriting semantics is presented in Fig. 3. It makes use of an equivalence relation $P_1 \equiv P_2$, defined as in [8] and saying that P_1 and P_2 are equal modulo "syntactic rearrangement" (we have, e.g., that $P \mid \mathbf{0} \equiv P$ and $P \mid Q \equiv Q \mid P$). The rule (Red Comm Input b) expresses that a process receives input from its child b, similarly for the other directional communication rules.

$$
\begin{array}{lll}
b[\mathsf{in}\ a.P \mid Q] \mid a[R] & \longrightarrow a[b[P \mid Q] \mid R] & (\mathsf{Red\ In}) \\
a[b[\mathsf{out}\ a.P \mid Q] \mid R] & \longrightarrow b[P \mid Q] \mid a[R] & (\mathsf{Red\ Out}) \\
\mathsf{open}\ a.P \mid a[Q] & \longrightarrow P \mid Q & (\mathsf{Red\ Open}) \\
(\vec{a})^{\star}.P \mid \langle \vec{M} \rangle^{\star}.Q & \longrightarrow P[\vec{a} := \vec{M}] \mid Q & (\mathsf{Red\ Comm}) \\
(\vec{a})^{\downarrow b}.P \mid b[\langle \vec{M} \rangle^{\star}.Q \mid R] & \longrightarrow P[\vec{a} := \vec{M}] \mid b[Q \mid R] & (\mathsf{Red\ Comm\ Input}\ b) \\
(\vec{a})^{\star}.P \mid b[\langle \vec{M} \rangle^{\uparrow}.Q \mid R] & \longrightarrow P[\vec{a} := \vec{M}] \mid b[Q \mid R] & (\mathsf{Red\ Comm\ Output}\ \uparrow) \\
b[(\vec{a})^{\star}.P \mid R] \mid \langle \vec{M} \rangle^{\downarrow b}.Q & \longrightarrow b[P[\vec{a} := \vec{M}] \mid R] \mid Q & (\mathsf{Red\ Comm\ Output}\ b) \\
b[(\vec{a})^{\uparrow}.P \mid R] \mid \langle \vec{M} \rangle^{\star}.Q & \longrightarrow b[P[\vec{a} := \vec{M}] \mid R] \mid Q & (\mathsf{Red\ Comm\ Input}\ \uparrow)
\end{array}
$$

$$
\begin{array}{lll}
P \longrightarrow Q \Rightarrow (\nu a).P & \longrightarrow (\nu a).Q & (\mathsf{Red\ Res}) \\
P \longrightarrow Q \Rightarrow a[P] & \longrightarrow a[Q] & (\mathsf{Red\ Amb}) \\
P \longrightarrow Q \Rightarrow P \mid R & \longrightarrow Q \mid R & (\mathsf{Red\ Par}) \\
\left.\begin{array}{r} P' \equiv P \\ P \longrightarrow Q \\ Q \equiv Q' \end{array}\right\} \Rightarrow P' & \longrightarrow Q' & (\mathsf{Red}\ \equiv)
\end{array}
$$

Fig. 3. The Soft-boxed Ambient Calculus, operational semantics.

$$
\begin{array}{lll}
A \in \mathsf{NameSet} & = & \text{non-empty finite set of names} \\
\Lambda \in \mathsf{DirSet} & = & \text{non-empty finite set of directions} \\
U \in \mathsf{PreTyp} & ::= & a \mid \mathsf{in}\ A \mid \mathsf{out}\ A \mid \mathsf{open}\ A \\
& & \mid \mathsf{const} \mid \mathsf{capseq} \mid \mathbf{0} \mid U_1 \mid U_2 \\
& & \mid (\nu a).U \mid A[U] \mid (\vec{a})^{\Lambda} \rightarrow U \mid \langle \vec{U} \rangle^{\Lambda} \\
& & \mid X \mid \mathsf{letrec}\ \vec{X} = \vec{U}\ \mathsf{in}\ X
\end{array}
$$

Fig. 4. The term syntax of shape types.

3 Shape Types

As described in the Introduction, we shall need types of unbounded depth. Therefore, we consider types to be potentially infinite trees, subject to certain restrictions given in Defs. 3.3 and 3.18.

Definition 3.1. *A* raw type *is a (possibly infinite) tree denoted by a term from the syntax in Fig. 4. The tree denoted by a type is determined by unfolding all occurrences of* $\mathsf{letrec}\ \vec{X} = \vec{T}\ \mathsf{in}\ X$ *infinitely often. We do not distinguish between raw types that are equal except for the consistent renaming of bound names.* □

A raw type is thus a tree built from *Parallel nodes*, labeled $\mid$ and with two children; *Nu nodes*, labeled $(\nu a).$ and with one child; *Ambient nodes*, labeled $A[]$ and with one child; *Input Nodes*, labeled $(\vec{a})^{\Lambda} \rightarrow$ and with one child; *Output Nodes*, labeled $\langle\rangle^{\Lambda}$ and with a number of children dependent on the arity of the output; and *Constant nodes*, forming the leaves of the tree. The type capseq will appear in the type for an expression of the form $M_1.M_2$ or ϵ; more about that later. The type const is the type of constants; it could be refined into several subtypes such as int and bool.

Definition 3.2. *A message type W is a raw type which is finite and where only Parallel nodes may occur as internal nodes. (That is, no Nu nodes, Ambient nodes, Input nodes, or Output nodes may occur.)* □

Definition 3.3. *A pre-type U is a raw type such that all infinite downward paths in U contains an infinite number of Ambient nodes, and such that all Output nodes in U are of the form $\langle \vec{W} \rangle^{\Lambda}$.* □

Trivially, a message type is also a pre-type. Thus process types are much like processes, except that

- as is standard in type systems, constants like 1, 2, 3 are replaced by const;
- in order to make the analysis more tractable (at the cost of losing precision), we replace sequential composition of capabilities and outputs with parallel composition (while still letting input operations be sequential);
- in order to type processes like $!a[!\mathsf{in}\ a]$, we allow types to have infinite depth;
- we may approximate names occurring in ambient position (so $\mathsf{in}\ \{a, b\}$ approximates $\mathsf{in}\ a$). This is to ensure that if a name in $\mathsf{fan}(U)$ is substituted by a set of names, then the resulting type is still valid. As syntactic sugar, we shall write $\mathsf{in}\ a$ for $\mathsf{in}\ \{a\}$, etc.

Moreover, as we shall see shortly (Fig. 6),

- parallel composition $|$ is idempotent (TypEq ParIdem), that is we do not keep track of multiplicities which are therefore potentially unbounded so there is no need for a replication operator on types;
- we collapse all occurrences of the same ambient of the same label (TypEq ParAmb), in order to keep the analysis manageable.

We could also have collapsed inputs with same arity, that is considered $(\vec{a}) \to U_1 \mid (\vec{a}) \to U_2$ to be equivalent to $(\vec{a}) \to (U_1 \mid U_2)$, but we chose not to.

While we shall often write (pre-)types using the term syntax given in Fig. 4, keep in mind that they are really trees. This is essentially the "equirecursive" approach, cf. the taxonomy in [12]. We do, however, only consider infinite trees that can be given a term representation. As shown in [12], in the presence of binders (like $(\nu a).T$) it is non-trivial to come up with an a condition on trees that corresponds exactly to the property "being representable by a pre-type term" (or "being representable by a tree automaton"). On the other hand, not all terms generate pre-types; a sufficient condition is that *(i)* the term contains no free recursion variables; and *(ii)* all "recursive calls" are *guarded*, that is beneath an Ambient node. For example, the term $\mathsf{letrec}\ X = (\nu a).\mathsf{in}\ a \mid X\ \mathsf{in}\ X$ is illegal as the corresponding tree looks like $(\nu a).\mathsf{in}\ a \mid (\nu a).\mathsf{in}\ a \mid (\nu a)\ldots$ which does not satisfy our requirement about infinite paths having Ambient nodes. On the other hand, the term $\mathsf{letrec}\ X_1 = X_2,\ X_2 = a[\mathsf{in}\ b \mid X_1]\ \mathsf{in}\ X_1$ is legal and corresponds to the pre-type $a[\mathsf{in}\ b \mid a[\mathsf{in}\ b \mid a[\ldots]]]$.

Also for pre-types we define the set of free names (occurring in ambient position):

$\mathsf{fn}(a)$	$= \{a\}$	$\mathsf{fan}(a)$	$= \emptyset$
for $f \in \{\mathsf{fn}, \mathsf{fan}\}$:			
$f(\Lambda)$	$= \{\, a \mid \downarrow a \in \Lambda \,\}$	$f(\mathsf{in}\ A)$	$= A$
$f(\mathsf{out}\ A)$	$= A$	$f(\mathsf{open}\ A)$	$= A$
$f(\mathsf{const})$	$= \emptyset$	$f(\mathsf{capseq})$	$= \emptyset$
$f(\mathbf{0})$	$= \emptyset$	$f(U_1 \mid U_2)$	$= f(U_1) \cup f(U_2)$
$f((\nu a).U)$	$= f(U) \setminus \{a\}$	$f((\vec{a})^{\Lambda} \rightarrow U)$	$= (f(U) \setminus \{\vec{a}\}) \cup f(\Lambda)$
$f(A[U])$	$= A \cup f(U)$	$f(\langle \vec{U} \rangle^{\Lambda})$	$= f(U) \cup f(\Lambda)$

Fig. 5. Free names and free ambient names for pre-types.

Definition 3.4. *The functions* $\mathsf{fn}(U)$ *and* $\mathsf{fan}(U)$ *are the least*[1] *total functions enjoying the properties listed in Fig. 5.* □

We employ a notion of equivalence for pre-types, with the aim that the equivalence $\doteq$ should satisfy $\doteq \Rightarrow^C \doteq$ where $\Rightarrow^C$ is defined in Fig. 6. In the presence of infinite types, however, we have to be a bit elaborate and essentially take a co-inductive approach:

Definition 3.5. *We say that* $U_1 \doteq U_2$ *iff* $U_1 \doteq_i U_2$ *holds for all* $i \geq 0$.
Here, the relation $U_1 \doteq_i U_2$ *(*$i \geq 0$*) is defined as follows:*

- $U_1 \doteq_0 U_2$ *always holds;*
- *for* $i \geq 1$, $\doteq_i$ *is the least relation* ϕ *satisfying* $\doteq_{i-1} \Rightarrow^C \phi$ *where the predicate* $\Rightarrow^C$ *is defined in Fig 6.* □

Lemma 3.6. *The relation* $\doteq$ *satisfies* $\doteq \Rightarrow^C \doteq$. □

It turns out that $\doteq$ is not the least relation ϕ such that $\phi \Rightarrow^C \phi$. We conjecture that adding one specific rule to Fig. 6 would make this hold, but we have not yet proven this. Fortunately, we do not need it.

Lemma 3.7. *If* a *is not free in* U*, then* $(\nu a).U \doteq U$. □

Proof. $(\nu a).U \doteq (\nu a).(U \mid \mathbf{0}) \doteq U \mid (\nu a).\mathbf{0} \doteq U \mid \mathbf{0} \doteq U$. □

Lemma 3.8. *If* $U_1 \doteq U_2$ *then* $\mathsf{fan}(U_1) = \mathsf{fan}(U_2)$ *and* $\mathsf{fn}(U_1) = \mathsf{fn}(U_2)$. □

3.1 Substitutions on Pre-types. We shall now define a notion of substitutions on pre-types. Just as for processes, substitution may not always be defined. But one can always substitute a *name type* for a name.

Definition 3.9. *A pre-type* U *is a name type iff* U *is a finite tree whose leaves are all names and whose remaining nodes are all Parallel nodes. For a name type* U*, we let* $\mathsf{names}(U)$ *denote the set of names occurring in* U. □

Note that a name type is also a message type.

$\phi_0 \Rightarrow^C \phi$ holds iff

	U	$\phi\ U$	(TypEq Refl)
$U_1\ \phi\ U_2 \Rightarrow$	U_2	$\phi\ U_1$	(TypEq Symm)
$U_1\ \phi\ U_2$ and $U_2\ \phi\ U_3 \Rightarrow$	U_1	$\phi\ U_3$	(TypEq Trans)
$U_1\ \phi\ U_2 \Rightarrow$	$(\nu a).U_1$	$\phi\ (\nu a).U_2$	(TypEq Res)
$U_1\ \phi\ U_2 \Rightarrow$	$U_1 \mid U$	$\phi\ U_2 \mid U$	(TypEq Par)
$U_1\ \phi_0\ U_2 \Rightarrow$	$a[U_1]$	$\phi\ a[U_2]$	(TypEq Amb)
$U_1\ \phi\ U_2 \Rightarrow$	$(\vec{a})^\lambda \rightarrow U_1$	$\phi\ (\vec{a})^\lambda \rightarrow U_2$	(TypEq Input)
$W_1\ \phi\ W_2 \Rightarrow$	$\langle \vec{W_1} \rangle^\lambda$	$\phi\ \langle \vec{W_2} \rangle^\lambda$	(TypEq Output)
	$U \mid \mathbf{0}$	$\phi\ U$	(TypEq ZeroPar)
	$U \mid U$	$\phi\ U$	(TypEq ParIdem)
	$U_1 \mid U_2$	$\phi\ U_2 \mid U_1$	(TypEq ParComm)
	$(U_1 \mid U_2) \mid U_3$	$\phi\ U_1 \mid (U_2 \mid U_3)$	(TypEq ParAssoc)
	$\mathsf{in}\ (A_1 \cup A_2)$	$\phi\ \mathsf{in}\ A_1 \mid \mathsf{in}\ A_2$	(TypEq InList)
	$\mathsf{out}\ (A_1 \cup A_2)$	$\phi\ \mathsf{out}\ A_1 \mid \mathsf{out}\ A_2$	(TypEq OutList)
	$\mathsf{open}\ (A_1 \cup A_2)$	$\phi\ \mathsf{open}\ A_1 \mid \mathsf{open}\ A_2$	(TypEq OpenList)
	$(A_1 \cup A_2)[U]$	$\phi\ A_1[U] \mid A_2[U]$	(TypEq AmbList)
	$(\vec{a})^{A_1 \cup A_2} \rightarrow U$	$\phi\ (\vec{a})^{A_1} \rightarrow U \mid (\vec{a})^{A_2} \rightarrow U$	(TypEq InputList)
	$\langle \vec{U} \rangle^{A_1 \cup A_2}$	$\phi\ \langle \vec{U} \rangle^{A_1} \mid \langle \vec{U} \rangle^{A_2}$	(TypEq OutputList)
	$a[U_1 \mid U_2]$	$\phi\ a[U_1] \mid a[U_2]$	(TypEq ParAmb)
	$(\nu a).(\nu b).U$	$\phi\ (\nu b).(\nu a).U$	(TypEq ResRes)
$a \notin \mathsf{fn}(U_1) \Rightarrow$	$(\nu a).(U_1 \mid U)$	$\phi\ U_1 \mid (\nu a).U$	(TypEq ResPar)
$a \neq b \Rightarrow$	$(\nu a).b[U]$	$\phi\ b[(\nu a).U]$	(TypEq ResAmb)
	$(\nu a).\mathbf{0}$	$\phi\ \mathbf{0}$	(TypEq ZeroRes)

Fig. 6. Congruence for pre-types.

Definition 3.10. *The result of substituting the message type W for the name a in the pre-type U, written $U[a := W]$ is defined in Fig. 7.* □

Lemma 3.11. *$U[a := W]$ is well-defined iff $a \in \mathsf{fan}(U)$ implies W is a name type.* □

Lemma 3.12. *Assume that $U_1 \doteq U_2$. Then $U_1[a := W]$ is well-defined iff $U_2[a := W]$ is well-defined, in which case $U_1[a := W] \doteq U_2[a := W]$.* □

3.2 Subtyping. There is an ordering $\leq$ on pre-types, with $U_1 \leq U_2$ meaning that U_1 is a more precise shape than U_2. Again, we must take a co-inductive approach:

Definition 3.13. *We say that $U_1 \leq U_2$ iff $U_1 \leq_i U_2$ holds for all $i \geq 0$.*

Here, the relation $U_1 \leq_i U_2$ ($i \geq 0$) is defined as follows:

- *$U_1 \leq_0 U_2$ always holds;*

[1] Wrt. the ordering defined by $f_1 \leq f_2$ iff $f_1(U) \subseteq f_2(U)$ for all U.

$$
\begin{array}{lll}
a[a := W] & = W & \\
b[a := W] & = b & \text{if } a \neq b \\
A[a := W] & = A & \text{if } a \notin A \\
A[a := W] & = (A \setminus \{a\}) \cup \mathsf{names}(W) & \text{if } a \in A,\ W \text{ a name type} \\
\Lambda[a := W] & = \Lambda & \text{if } a \notin \mathsf{fn}(\Lambda) \\
\Lambda[a := W] & = (\Lambda \setminus \{\downarrow a\}) \cup \{\downarrow b \mid b \in \mathsf{names}(W)\} & \text{if } a \in \mathsf{fn}(\Lambda),\ W \text{ a name type} \\
(\mathsf{in}\ A)[a := W] & = \mathsf{in}\ (A[a := W]) & \\
(\mathsf{out}\ A)[a := W] & = \mathsf{out}\ (A[a := W]) & \\
(\mathsf{open}\ A)[a := W] & = \mathsf{open}\ (A[a := W]) & \\
\mathsf{const}[a := W] & = \mathsf{const} & \\
\mathsf{capseq}[a := W] & = \mathsf{capseq} & \\
\mathbf{0}[a := W] & = \mathbf{0} & \\
(U_1 \mid U_2)[a := W] & = (U_1[a := W]) \mid (U_2[a := W]) & \\
((\nu b).U)[a := W] & = (\nu b).(U[a := W]) & \text{if } a \neq b, b \notin \mathsf{fn}(W) \\
(A[U])[a := W] & = (A[a := W])[U[a := W]] & \\
((\vec{a})^{\Lambda} \to U)[a := W] & = (\vec{a})^{\Lambda[a:=W]} \to U[a := W] & \text{if } \{\vec{a}\} \cap (\{a\} \cup \mathsf{fn}(W)) = \emptyset \\
(\langle \vec{W} \rangle^{\Lambda})[a := W] & = \langle \vec{W}[a := W] \rangle^{\Lambda[a:=W]} & \\
X[a := W] & = X &
\end{array}
$$

$$(\mathsf{letrec}\ \vec{X} = \vec{U}\ \mathsf{in}\ X)[a := W] = \mathsf{letrec}\ \vec{X} = \vec{U}[a := W]\ \mathsf{in}\ X$$

Fig. 7. Substitution on pre-types.

$\phi_0 \Rightarrow^S \phi$ holds iff

$$
\begin{array}{rll}
\mathbf{0} & \phi\ U & (\mathsf{TypSub\ Zero}) \\
U_1 \doteq U_2 \Rightarrow U_1 & \phi\ U_2 & (\mathsf{TypSub\ Refl}) \\
U_1\ \phi\ U_2 \text{ and } U_2\ \phi\ U_3 \Rightarrow U_1 & \phi\ U_3 & (\mathsf{TypSub\ Trans}) \\
U_1\ \phi\ U_2 \Rightarrow (\nu a).U_1 & \phi\ (\nu a).U_2 & (\mathsf{TypSub\ Res}) \\
U_1\ \phi\ U_2 \Rightarrow U_1 \mid U & \phi\ U_2 \mid U & (\mathsf{TypSub\ Par}) \\
U_1\ \phi_0\ U_2 \Rightarrow a[U_1] & \phi\ a[U_2] & (\mathsf{TypSub\ Amb}) \\
U_1\ \phi\ U_2 \Rightarrow (\vec{a})^{\lambda} \to U_1 & \phi\ (\vec{a})^{\lambda} \to U_2 & (\mathsf{TypSub\ Input}) \\
\vec{W}_1\ \phi\ \vec{W}_2 \Rightarrow \langle \vec{W}_1 \rangle^{\lambda} & \phi\ \langle \vec{W}_2 \rangle^{\lambda} & (\mathsf{TypSub\ Output})
\end{array}
$$

Fig. 8. Subtyping.

– *for $i \geq 1$, $\leq_i$ is the least relation ϕ satisfying $\leq_{i-1} \Rightarrow^S \phi$ where the predicate $\Rightarrow^S$ is defined in Fig 8.* □

Note that subtyping is covariant in the output position (as in [20]), whereas the input position in the dependent types of PolyA is a list of names and is thus essentially invariant.

Lemma 3.14. *The relation $U_1 \leq U_2$ satisfies $\leq \Rightarrow^S \leq$.* □

It turns out that $U_1 \leq U_2$ is not the least relation ϕ such that $\phi \Rightarrow^S \phi$, but fortunately we do not need this.

Lemma 3.15. *Parallel composition* $|$ *is least upper bound on pre-types wrt. the ordering* $\leq$. □

The operator $|$ might thus also be viewed as a disjunction operator.

Lemma 3.16. *If* $U_1 \leq U_2$ *then* $\mathsf{fan}(U_1) \subseteq \mathsf{fan}(U_2)$ *and* $\mathsf{fn}(U_1) \subseteq \mathsf{fn}(U_2)$. □

Note that if we had not been careful w.r.t. the use of i versus $i-1$ when defining $U_1 \leq_i U_2$, we might have ended up with an inconsistent relation.

Lemma 3.17. *Assume that* $U_1 \leq U_2$. *If* $U_2[a := W]$ *is well-defined, also* $U_1[a := W]$ *is well-defined, and* $U_1[a := W] \leq U_2[a := W]$. □

3.3 Closedness. We demand that types are closed under certain rules that estimate the possible future states of a process of the type.

Definition 3.18. *A type* T *is a pre-type that is* closed, *where the predicate "closed" is the largest predicate satisfying that if* T *is closed then the clauses in Fig. 9 hold.* □

This is well-defined, since Fig. 9 does in fact define a motonone operator on sets of types.

We could replace the rule (Clos In) by

$$a[U_1] \mid b[U_2] \leq T \text{ and } (\mathsf{in}\ b) \leq U_1 \Rightarrow b[a[U_1]] \leq T$$

but it is worth noticing that the reason why subject reduction would still work is that we have the rule (TypEq ParAmb). Alternatively, we could replace (Clos In) by the rule

$$a[U_1] \leq T \text{ and } (\mathsf{in}\ b) \leq U_1 \Rightarrow b[a[U_1]] \leq T$$

but then computing the closure would create far too many configurations.

4 The PolyA System

In Fig. 10 we define two typing judgements $M : W$ and $P : U$. Most rules are very straightforward; the rule (Proc Repl) reflects that we do not count multiplicities.

We have the following result which is crucial for our later soundness results.

Lemma 4.1. *Assume that* $M : W$ *with* W *a name type. Then* M *is a name.* □

Note that if it wasn't for the presence of capseq in the rules (Exp ϵ) and (Exp Action), this result would not hold: we would have $a.a : a \mid a$ and therefore (by idempotency of $|$) $a.a : a$; similarly, we would have $\epsilon : \mathbf{0}$ and therefore also $\epsilon : a$. Spurious occurrences of capseq are eliminated in (Proc Action).

Lemma 4.2. *Assume that* $P : U$, *that* $M : W$, *and that* $U[a := W]$ *is well-defined. Then also* $P[a := M]$ *is well-defined, and* $P[a := M] : U[a := W]$. □

T is closed iff

$$\left.\begin{array}{r} a[U_1] \mid b[U_2] \leq T \\ (\mathsf{in}\ b) \leq U_1 \end{array}\right\} \Rightarrow b[a[U_1] \mid U_2] \leq T \qquad (\mathsf{Clos\ In})$$

$$\left.\begin{array}{r} b[a[U_1]] \leq T \\ (\mathsf{out}\ b) \leq U_1 \end{array}\right\} \Rightarrow a[U_1] \leq T \qquad (\mathsf{Clos\ Out})$$

$$\left.\begin{array}{r} a[U] \leq T \\ (\mathsf{open}\ a) \leq T \end{array}\right\} \Rightarrow U \leq T \qquad (\mathsf{Clos\ Open})$$

$$\left.\begin{array}{r} (\vec{a})^\star \rightarrow U \leq T \\ \langle \vec{W} \rangle^\star \leq T \end{array}\right\} \Rightarrow \left\{\begin{array}{l} U[\vec{a} := \vec{W}] \text{ well-defined} \\ U[\vec{a} := \vec{W}] \leq T \end{array}\right. \qquad (\mathsf{Clos\ Comm})$$

$$\left.\begin{array}{r} (\vec{a})^{\downarrow b} \rightarrow U \leq T \\ b[U'] \leq T \\ \langle \vec{W} \rangle^\star \leq U' \end{array}\right\} \Rightarrow \left\{\begin{array}{l} U[\vec{a} := \vec{W}] \text{ well-defined} \\ U[\vec{a} := \vec{W}] \leq T \end{array}\right. \qquad (\mathsf{Clos\ Comm\ Input}\ b)$$

$$\left.\begin{array}{r} (\vec{a})^\star \rightarrow U \leq T \\ b[U'] \leq T \\ \langle \vec{W} \rangle^\uparrow \leq U' \end{array}\right\} \Rightarrow \left\{\begin{array}{l} U[\vec{a} := \vec{W}] \text{ well-defined} \\ U[\vec{a} := \vec{W}] \leq T \end{array}\right. \qquad (\mathsf{Clos\ Comm\ Output}\ \uparrow)$$

$$\left.\begin{array}{r} \langle \vec{W} \rangle^{\downarrow b} \leq T \\ b[U'] \leq T \\ (\vec{a})^\star \rightarrow U \leq U' \end{array}\right\} \Rightarrow \left\{\begin{array}{l} U[\vec{a} := \vec{W}] \text{ well-defined} \\ U[\vec{a} := \vec{W}] \leq U' \end{array}\right. \qquad (\mathsf{Clos\ Comm\ Output}\ b)$$

$$\left.\begin{array}{r} \langle \vec{W} \rangle^\star \leq T \\ b[U'] \leq T \\ (\vec{a})^\uparrow \rightarrow U \leq U' \end{array}\right\} \Rightarrow \left\{\begin{array}{l} U[\vec{a} := \vec{W}] \text{ well-defined} \\ U[\vec{a} := \vec{W}] \leq U' \end{array}\right. \qquad (\mathsf{Clos\ Comm\ Input}\ \uparrow)$$

$$a[U_1] \leq T \Rightarrow \exists \text{ closed } T_1 : \left\{\begin{array}{l} U_1 \leq T_1 \\ a[T_1] \leq T \end{array}\right. \qquad (\mathsf{Clos\ Amb})$$

$$(\nu a).U_1 \leq T \Rightarrow \exists \text{ closed } T_1 : \left\{\begin{array}{l} U_1 \leq T_1 \\ (\nu a).T_1 \leq T \end{array}\right. \qquad (\mathsf{Clos\ Res})$$

In the 5 rules for communications, an extra condition is that the arities match.

Fig. 9. Conditions for closedness.

Not all processes can be assigned a closed type T, an example is $(a)^\star.a[\mathbf{0}] \mid \langle \mathsf{in}\ b \rangle^\star$. If there exists a type T with $P : T$, however, then P is semantically sound as shown by the following two theorems:

Theorem 4.3 (Subject reduction). *If $P \longrightarrow Q$ and $P : T$ then $Q : T$.* □

The proof is by induction in the derivation of $P \longrightarrow Q$. An interesting case is where $P = (\nu n).P_1$ and $Q = (\nu n).Q_1$ and $P_1 \longrightarrow Q_1$. Then there exists U_1 with $(\nu n).U_1 \leq T$ such that $P_1 : U_1$. Since T is closed, there exists closed T_1 with $U_1 \leq T_1$ and $(\nu n).T_1 \leq T$. Since $P_1 : T_1$, inductively we have $Q_1 : T_1$. But this shows $Q : (\nu n).T_1$ and therefore $Q : T$, as desired.

$\dfrac{M : W \quad W \leq W'}{M : W'}$	(Exp Subsumpt)	$\dfrac{P : U \quad U \leq U'}{P : U'}$	(Proc Subsumpt)
$\dfrac{}{a : a}$	(Exp a)	$\dfrac{}{c : \mathsf{const}}$	(Exp Const)
$\dfrac{}{\epsilon : \mathsf{capseq}}$	(Exp ϵ)	$\dfrac{M_1 : W_1 \quad M_2 : W_2}{M_1.M_2 : W_1 \mid W_2 \mid \mathsf{capseq}}$	(Exp Action)
$\dfrac{}{\mathsf{in}\ a : \mathsf{in}\ a}$	(Exp In)	$\dfrac{}{\mathsf{out}\ a : \mathsf{out}\ a}$	(Exp Out)
$\dfrac{}{\mathsf{open}\ a : \mathsf{open}\ a}$	(Exp Open)	$\dfrac{}{\mathbf{0} : \mathbf{0}}$	(Proc Zero)
$\dfrac{P_1 : U_1 \quad P_2 : U_2}{P_1 \mid P_2 : U_1 \mid U_2}$	(Proc Par)	$\dfrac{P : U}{!P : U}$	(Proc Repl)
$\dfrac{M : W \mid \mathsf{capseq} \quad P : U}{M.P : W \mid U}$	(Proc Action)		
$\dfrac{P : U}{(\nu a).P : (\nu a).U}$	(Proc Res)	$\dfrac{P : U}{a[P] : a[U]}$	(Proc Amb)
$\dfrac{P : U}{(\vec{a})^\lambda.P : (\vec{a})^\lambda \to U}$	(Proc Input)	$\dfrac{M_1 : W_1 \ldots M_k : W_k \quad P : U}{\langle \vec{M} \rangle^\lambda.P : \langle \vec{W} \rangle^\lambda \mid U}$	(Proc Output)

Fig. 10. Our type system PolyA.

Theorem 4.4 (Safety). *If $P : T$ then execution of P will never give rise to an ill-defined substitution.* □

4.1 Principality The following result, though of limited interest since U_P is not necessarily closed, is immediate:

Theorem 4.5 (Principality, naïve). *Given a process P, there exists a pre-type U_P such that*

- *$P : U_P$;*
- *if $P : U$ then $U_P \leq U$.* □

On the other hand, what we would like to have is

Conjecture 4.6 (Principality, hoped for). Given a process P that can be assigned a *type*, there exists a type T_P such that

- $P : T_P$;
- T_P is closed;
- if $P : T$ then $T_P \leq T$. □

We do not know whether the above holds. We have, however, been able to prove the following more restricted version:

Theorem 4.7 (Principality, limited). *Assume that we are working in a restriction of* PolyA *which forbids ν and type recursion. Given a process P that can be assigned a type, there exists a type T_P such that*

– $P : T_P$;
– T_P *is closed;*
– *if* $P : T$ *then* $T_P \leq T$.

□

To prove this theorem, we need the following result:

Lemma 4.8. *Given a non-empty set* $\{U_i \mid i \in I\}$ *of finite,* ν*-free pre-types. Then one can construct a (finite and* ν*-free) pre-type* U *which is the greatest lower bound of* $\{U_i \mid i \in I\}$*. Moreover, if all* U_i *are closed also* U *is closed.* □

4.2 An Anomaly with open. Consider this term:

$$a[\mathsf{in}\ b.\mathbf{0}] \mid b[\mathsf{open}\ a.\mathbf{0}]$$

Ignoring the closedness requirement, we could give it this pre-type:

$$a[\mathsf{in}\ b] \mid b[\mathsf{open}\ a]$$

To close this pre-type, observe that a can go into b:

$$a[\mathsf{in}\ b] \mid b[a[\mathsf{in}\ b] \mid \mathsf{open}\ a]$$

Then a can be opened:

$$a[\mathsf{in}\ b] \mid b[a[\mathsf{in}\ b] \mid \mathsf{open}\ a \mid \mathsf{in}\ b]$$

Now, one copy of b can go into another:

$$\ldots \mid b[\ldots \mid \mathsf{in}\ b \mid b[\ldots \mid \mathsf{in}\ b]]$$

This repeats forever. To close the pre-type requires a recursive type:

$$a[\mathsf{in}\ b] \mid \mathsf{letrec}\ X = b[a[\mathsf{in}\ b] \mid \mathsf{open}\ a \mid \mathsf{in}\ b \mid X]\ \mathsf{in}\ X$$

Below, we discuss some ways to deal with this anomaly.

– One could just try to live with it. The types would look ugly, but it seems that many safe programs using open would be typable and hence our system would prove them safe, despite the anomaly. This seems to represent an improvement over previous type systems.
– One could avoid using open, thus sticking to the core BA calculus. However, at some point a type system handling open flexibly would be desirable.[2]
– One could attempt to add multiplicities to the types. This would have worked for our motivating example, ensuring that the nested a ambient did not have an in b capability. But in general, counting is not enough because the types "confuse the past with the future", e.g., the count z must be ω to make this type closed:

$$(\mathsf{open}\ a)^1 \mid a[(\mathsf{in}\ b)^1]^1 \mid (\mathsf{in}\ b)^z$$

– One could introduce union types. Theoretically, these could work, but they are not feasible because each point in the possible future state space would likely become a separate position in the type.

[2] E.g., for new applications in modeling intracellular biological processes see `http://www.wisdom.weizmann.ac.il/~aviv/`.

5 Embedding Standard Ambient Types

It can be shown that the type system of Cardelli and Gordon [8] can be embedded into PolyA. For ν-free programs, we know how to develop a mechanical translation from typing derivations of their system. We illustrate the idea with an example and leave the formal details to future work. Consider the process

$$P = a[\mathsf{open}\ b.(y)^\star.\mathsf{in}\ y.\mathbf{0} \mid b[\langle c\rangle^\star.\mathbf{0}]] \mid c[\mathbf{0}]$$

which can be typed in the system of [8], by assigning the following types to the (free and bound) names:

$$\begin{aligned} c, y : T_1 &= \mathsf{Amb}[\mathsf{shh}] \\ a, b : T_2 &= \mathsf{Amb}[T_1] \end{aligned}$$

A mechanical translation converts this assignment into the PolyA type for P given below:

$$\begin{aligned} \mathsf{letrec}\ X_C &= \mathsf{in}\ \{a,b,c,y\} \mid \mathsf{out}\ \{a,b,c,y\} \mid \mathsf{capseq} \\ X_A &= a[X_2] \mid b[X_2] \mid c[X_1] \\ X_1 &= X_A \mid X_C \mid \mathsf{open}\ \{c,y\} \\ X_2 &= X_A \mid X_C \mid \mathsf{open}\ \{a,b\} \mid \langle c\rangle^\star \mid \langle y\rangle^\star \mid (y)^\star \to X_2 \\ X &= (\nu y).X_1 \\ \mathsf{in}\ \ X & \end{aligned}$$

The intuition is that while no ambient movements are ruled out, an ambient is only allowed to dissolve ambients with the same type, and of course only allowed to output and input entities of the appropriate type. For example, inside c we can (X_1) have other ambients (X_A), have arbitrary in and out capabilities (X_C), open other copies of c as well as open y, but since c has type Amb[shh] no input or output actions are allowed.

We expect that a similar embedding can be done for the mobility type system of [6]. The translation would then give P a PolyA type which is more refined than the one above, but still much coarser than the best PolyA type for P which is:

$$\begin{aligned} \mathsf{letrec}\ X_a &= \mathsf{open}\ b \mid \mathsf{in}\ c \mid \langle c\rangle^\star \mid \mid b[\langle c\rangle^\star] \mid (y)^\star \to \mathsf{in}\ y \\ X &= a[X_a] \mid c[a[X_a]] \\ \mathsf{in}\ \ X & \end{aligned}$$

6 Other Kinds of Poly-morphic/variant Analysis

Several other papers have explored the idea of letting the analysis of an ambient subprocess depend on its possible contexts—a task which requires an estimate of the possible shapes of the ambient tree structure. Below we shall list a few. None of these handle communication, however, so none can prove the safety of our example polymorphic messenger from (1) in Sec. 1.

Shape grammars. In [17], an analysis is developed which returns a set of grammars such that at any step, the current process can be described by one of these grammars. The analysis is very precise, but potentially also very expensive.

Kleene Analysis. In [16], a 3-valued logic is used to estimate the possible shapes. The framework allows for trade-offs w.r.t. precision vs. costs.

Abstract Interpretation. The system of [13] keeps track of the context "one level up". This is sufficient to achieve a quite precise analysis, yet is "only" polynomial (n^7).

7 Conclusion

We have presented a type system for a variant of the ambient calculus. The system is poly-morphic/variant in that an ambient is analyzed differently for different interactions it enters into, and has dependent typing where the analysis tracks which values are communicated and reacts accordingly. It can prove the safety of generic mobile agents that no previous type system for the ambient calculus can type.

Future work includes:

- Further investigating the relationship to other systems.
- Writing a type inference algorithm, the bulk of which is the construction of an algorithm for computing the closure of a type (Sect. 3.3). In general, the naïve application of the rules in Fig. 9 will not terminate, so approximations are clearly needed. For instance, we may want to replace

  $$a[T_1 \mid a[T_2]]$$

  by the infinite type

  $$\mathsf{letrec}\ X = a[X \mid T_1 \mid T_2]\ \mathsf{in}\ X$$

  We expect that further inspiration may be possible from the literature on "widening" [10].
- Evaluating the practical usefulness and feasibility of our approach.

References

[1] T. Amtoft, A. J. Kfoury, S. M. Pericas-Geertsen. What are polymorphically-typed ambients? Technical Report BUCS-TR-2000-021, Comp. Sci. Dept., Boston Univ., 2000.

[2] T. Amtoft, A. J. Kfoury, S. M. Pericas-Geertsen. What are polymorphically-typed ambients? In D. Sands, ed., *ESOP 2001, Genova*, vol. 2028 of *LNCS*. Springer-Verlag, 2001. An extended version appears as [1].

[3] T. Amtoft, A. J. Kfoury, S. M. Pericas-Geertsen. Orderly communication in the ambient calculus. *Computer Languages*, 2002. To appear.

[4] M. Bugliesi, G. Castagna, S. Crafa. Boxed ambients. In *4th International Conference on Theoretical Aspects of Computer Science (TACS'01)*, vol. 2215 of *LNCS*. Springer-Verlag, 2001.

[5] M. Bugliesi, S. Crafa, M. Merro, V. Sassone. Communication interference in mobile boxed ambients. In *FST & TCS 2002*, 2002.

[6] L. Cardelli, G. Ghelli, A. D. Gordon. Mobility types for mobile ambients. In J. Wiedermann, P. van Emde Boas, M. Nielsen, eds., *ICALP'99*, vol. 1644 of *LNCS*. Springer-Verlag, 1999. Extended version appears as Microsoft Research Technical Report MSR-TR-99-32, 1999.

[7] L. Cardelli, A. D. Gordon. Mobile ambients. In M. Nivat, ed., *FoSSaCS'98*, vol. 1378 of *LNCS*. Springer-Verlag, 1998.

[8] L. Cardelli, A. D. Gordon. Types for mobile ambients. In *POPL'99, San Antonio, Texas*. ACM Press, 1999.

[9] M. Coppo, M. Dezani-Ciancaglini. A fully abstract model for higher-order mobile ambients. In *VMCAI 2002*, vol. 2294 of *LNCS*, 2002.

[10] P. Cousot, R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Prog. Lang. Implementation & Logic Prog., 4th Int'l Symp.*, vol. 631 of *LNCS*. Springer-Verlag, 1992.

[11] C. Fournet, G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.

[12] N. Glew. A theory of second-order trees. In *ESOP 2002*, vol. 2305 of *LNCS*. Springer-Verlag, 2002.

[13] F. Levi, S. Maffeis. An abstract interpretation framework for analysing mobile ambients. In *SAS'01*, vol. 2126 of *LNCS*. Springer-Verlag, 2001.

[14] F. Levi, D. Sangiorgi. Controlling interference in ambients. In *POPL'00, Boston, Massachusetts*. ACM Press, 2000.

[15] R. Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge Press, 1999.

[16] F. Nielson, H. R. Nielson, M. Sagiv. A Kleene analysis of mobile ambients. In *Programming Languages & Systems, 9th European Symp. Programming*, vol. 1782 of *LNCS*. Springer-Verlag, 2000.

[17] H. R. Nielson, F. Nielson. Shape analysis for mobile ambients. In *POPL'00, Boston, Massachusetts*. ACM Press, 2000. A revised and extended version has appeared in Nordic Journal of Computing, 8:233–275, 2001.

[18] D. Teller, P. Zimmer, D. Hirschkoff. Using ambients to control resources. In *CONCUR'02*, vol. 2421 of *LNCS*. Springer-Verlag, 2002.

[19] J. Vitek, G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, vol. 1686 of *LNCS*. Springer-Verlag, 1999.

[20] P. Zimmer. Subtyping and typing algorithms for mobile ambients. In *FOSSACS 2000, Berlin*, vol. 1784 of *LNCS*. Springer-Verlag, 2000.