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 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. This system 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 [16] is probably still the most popular, the ambient calculus [8] has gained quite some popularity, with several other frameworks (to name one: the MR-calculus [13]) joining the competition. Each of these calculi involve the passing of names, so it seems that

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

in order to enable a precise analysis, some kind of dependent typing would be useful. We shall explore this idea in the present paper.

To have a concrete model, we choose a variant of Boxed Ambients [4] in which communication can take place not only between processes which are in the same ambient, but also between parents and their children. It should be easy to extend our work to the Seal calculus [20], 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 [8], 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 ambient named q is an example of a kind of generic functionality, namely a "messenger". That is, q goes to various places looking for messages to deliver, at these places 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.

Often, type systems for variants of the ambient calculus follow the example of Cardelli & Gordon's seminal system [9] 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 boxed ambients, 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:

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

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, 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. 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. Here is an example judgement:

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

The types say nothing about the number of copies of a feature at a location, unlike what is the case in [19]. Features of our systems are as follows:

– There are singleton types of ambient names and explicit dependencies on communication, as illustrated by the 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 are deliberately collapsing most sequencing information because increasing precision in the types by keeping track of the sequence of actions has already been explored in detail by Amtoft (one of this paper's authors), Kfoury, and Pericas [3, 2].

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

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

but do not do the same 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), though for messages only certain types are allowed.

– 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.

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

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

This proves that the example process is well behaved, something which no previous type system for ambients can do.

2 The Soft-boxed Ambient Calculus

Our soft-boxed ambient calculus is given in Fig. 1. It has the features of the boxed ambient calculus [4] and the original ambient calculus [8], because this

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

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

allows us to discuss issues of typing both with and without the open capability. 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 [15] and have since then become quite popular (e.g., they were added to boxed ambients in [5]), partly due to the fact that their presence often significantly improves the precision of an analysis. It is doubtful, however, whether they would benefit our purposes, for the following reasons. Co-capabilities may be used to say "we only allow someone in *after* something has happened", but in this paper our type system deliberately collapses information about sequencing other than input actions. Our type system could be easily 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 our type system 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 shall allow constants, as they are useful for writing examples; it is relatively straightforward also to allow operations on these. 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 "am-

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

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

bient naming position". A member of $\mathsf{fan}(P)$ can only be replaced by another name, not by an arbitrary expression. These functions are presented in Fig 2. Note that the only difference between $\mathsf{fan}()$ and $\mathsf{fn}()$ is in the case for $M = a$.

We write $X[a := M]$ for the process resulting from 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, ..., M_n respectively for a_1, ..., 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 $P[a := M]$ may not always be well-defined, as in $(a[Q])[a := \mathsf{in}\ b]$ which would seem to result in the ill-formed $(\mathsf{in}\ b)[Q]$. This is the only way in our system that a process can "go wrong". A major task of our type system is to ensure that all substitutions performed at runtime are well-defined, cf. rule (Red Comm) in Fig 3.

Lemma 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 [9] 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 $P \mid Q \equiv Q \mid P$).

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

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

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

$$
\begin{array}{lcl}
A \in \mathsf{NameSet} & = & \text{non-empty set of names} \\
\Lambda \in \mathsf{LocSet} & = & \text{non-empty set of directions} \\
T \in \mathsf{Typ} & ::= & a \mid \mathsf{in}\ A \mid \mathsf{out}\ A \mid \mathsf{open}\ A \\
 & & \mid \mathsf{const} \mid \mathsf{action} \mid \mathbf{0} \mid T_1 \mid T_2 \\
 & & \mid (\nu a).T \mid A[T] \mid (\vec{a})^\Lambda \to T \mid \langle \vec{T} \rangle^\Lambda \\
 & & \mid X \mid \mathsf{letrec}\ \vec{X} = \vec{T}\ \mathsf{in}\ X
\end{array}
$$

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 Def. 3.

Definition 1. *A* pre-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 pre-types that are equal except for the consistent renaming of bound names.* □

A pre-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 $(\vec{a})^\Lambda \to$ 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 action 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 2. *A message type U is a pre-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. *A type T is a pre-type such that all infinite downward paths in T contains an infinite number of nodes that are either Ambient nodes or Input nodes, and such that all Output nodes in T are of the form $\langle \vec{U} \rangle^{\Lambda}$.* □

Thus process types are much like processes, except that

- as is standard in type systems, constants like 1, 2, 3 are replaced by const;
- we may approximate names occurring[1] in ambient position (so $\mathsf{in}\ \{a, b\}$ approximates $\mathsf{in}\ a$);
- 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.

Moreover, as we shall see shortly (Fig. 7),

- parallel composition $\mid$ is idempotent ($\mathsf{TypEq\ ParIdem}$), 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 ($\mathsf{TypEq\ ParAmb}$), in order to keep the analysis manageable.

We could also have collapsed inputs with same arity, that is considered $(\vec{a}) \to T_1 \mid (\vec{a}) \to T_2$ to be equivalent to $(\vec{a}) \to (T_1 \mid T_2)$, but we chose not to.

While we shall often write types using the term syntax given in Fig. 4, keep in mind that types 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 type term" (or "being representable by a tree automaton"). On the other hand, not all terms generate 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 or an Input 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 or Input 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 type $a[\mathsf{in}\ b \mid a[\mathsf{in}\ b \mid a[\ldots]]]$.

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

Definition 4. *The functions $\mathsf{fn}(T)$ and $\mathsf{fan}(T)$ are the least[2] total functions enjoying the properties listed in Fig. 5.* □

[1] This is to ensure that if a name in $\mathsf{fan}(T)$ is substituted by a set of names, then the resulting type is still valid.

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

$\mathsf{fn}(\Lambda)$	$= \{\, a \mid \downarrow a \in \Lambda \,\}$	$\mathsf{fan}(\Lambda)$	$= \{\, a \mid \downarrow a \in \Lambda \,\}$
$\mathsf{fn}(a)$	$= \{a\}$	$\mathsf{fan}(a)$	$= \emptyset$
$\mathsf{fn}(\mathsf{in}\ A)$	$= A$	$\mathsf{fan}(\mathsf{in}\ A)$	$= A$
$\mathsf{fn}(\mathsf{out}\ A)$	$= A$	$\mathsf{fan}(\mathsf{out}\ A)$	$= A$
$\mathsf{fn}(\mathsf{open}\ A)$	$= A$	$\mathsf{fan}(\mathsf{open}\ A)$	$= A$
$\mathsf{fn}(\mathsf{const})$	$= \emptyset$	$\mathsf{fan}(\mathsf{const})$	$= \emptyset$
$\mathsf{fn}(\mathsf{action})$	$= \emptyset$	$\mathsf{fan}(\mathsf{action})$	$= \emptyset$
$\mathsf{fn}(\mathbf{0})$	$= \emptyset$	$\mathsf{fan}(\mathbf{0})$	$= \emptyset$
$\mathsf{fn}(T_1 \mid T_2)$	$= \mathsf{fn}(T_1) \cup \mathsf{fn}(T_2)$	$\mathsf{fan}(T_1 \mid T_2)$	$= \mathsf{fan}(T_1) \cup \mathsf{fan}(T_2)$
$\mathsf{fn}((\nu a).T)$	$= \mathsf{fn}(T) \setminus \{a\}$	$\mathsf{fan}((\nu a).T)$	$= \mathsf{fan}(T) \setminus \{a\}$
$\mathsf{fn}(A[T])$	$= A \cup \mathsf{fn}(T)$	$\mathsf{fan}(A[T])$	$= A \cup \mathsf{fan}(T)$
$\mathsf{fn}((\vec{a})^{\Lambda} \to T)$	$= (\mathsf{fn}(T) \setminus \{\vec{a}\}) \cup \mathsf{fn}(\Lambda)$	$\mathsf{fan}((\vec{a})^{\Lambda} \to T)$	$= (\mathsf{fan}(T) \setminus \{\vec{a}\}) \cup \mathsf{fan}(\Lambda)$
$\mathsf{fn}(\langle \vec{T} \rangle^{\Lambda})$	$= \mathsf{fn}(T) \cup \mathsf{fn}(\Lambda)$	$\mathsf{fan}(\langle \vec{T} \rangle^{\Lambda})$	$= \mathsf{fan}(T) \cup \mathsf{fan}(\Lambda)$

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

Lemma 2. *For all T we have $\mathsf{fan}(T) \subseteq \mathsf{fn}(T)$.* □

We employ a notion of equivalence for types, with the aim that the properties listed in Fig. 7 should hold. In the presence of infinite types, however, we have to be more elaborate:

Definition 5. *We say that $T_1 \doteq T_2$ iff $T_1 \doteq_i T_2$ holds for all $i \geq 0$, where $T_1 \doteq_0 T_2$ always holds and where for $i \geq 1$, $T_1 \doteq_i T_2$ is the least relation satisfying the clauses in Fig 6.* □

Lemma 3. *The relation $T_1 \doteq T_2$ enjoys the properties listed in Fig. 7.* □

It turns out that $T_1 \doteq T_2$ is not the least relation enjoying the properties of Fig. 7. We conjecture that adding one specific rule to Fig. 7 would make this hold, but we have not yet proven this. Fortunately, we do not need it.

Lemma 4. *If a is not free in T, then $(\nu a).T \doteq T$.* □

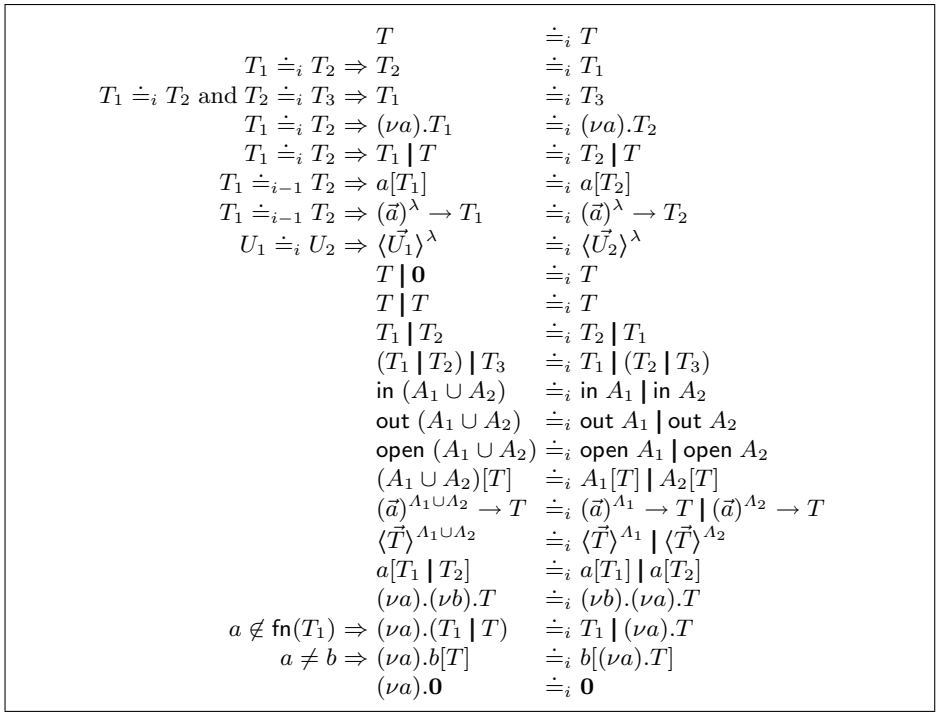
Proof. $(\nu a).T \doteq (\nu a).(T \mid \mathbf{0}) \doteq T \mid (\nu a).\mathbf{0} \doteq T \mid \mathbf{0} \doteq T$. □

Lemma 5. *If $T_1 \doteq T_2$ then $\mathsf{fan}(T_1) = \mathsf{fan}(T_2)$ and $\mathsf{fn}(T_1) = \mathsf{fn}(T_2)$.* □

Note that if we had not been careful w.r.t. the use of i versus $i - 1$ when defining $T_1 \doteq_i T_2$, we might have ended up with an inconsistent relation.

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

Definition 6. *A type T is a name type iff T is a finite tree whose leaves are all names and whose remaining nodes are all Parallel nodes. For a name type T, we let $\mathsf{names}(T)$ denote the set of names occurring in T.* □



$$
\begin{array}{rll}
 & T & \doteq_i T \\
T_1 \doteq_i T_2 \Rightarrow & T_2 & \doteq_i T_1 \\
T_1 \doteq_i T_2 \text{ and } T_2 \doteq_i T_3 \Rightarrow & T_1 & \doteq_i T_3 \\
T_1 \doteq_i T_2 \Rightarrow & (\nu a).T_1 & \doteq_i (\nu a).T_2 \\
T_1 \doteq_i T_2 \Rightarrow & T_1 \mid T & \doteq_i T_2 \mid T \\
T_1 \doteq_{i-1} T_2 \Rightarrow & a[T_1] & \doteq_i a[T_2] \\
T_1 \doteq_{i-1} T_2 \Rightarrow & (\vec{a})^{\lambda} \to T_1 & \doteq_i (\vec{a})^{\lambda} \to T_2 \\
U_1 \doteq_i U_2 \Rightarrow & \langle \vec{U_1} \rangle^{\lambda} & \doteq_i \langle \vec{U_2} \rangle^{\lambda} \\
 & T \mid \mathbf{0} & \doteq_i T \\
 & T \mid T & \doteq_i T \\
 & T_1 \mid T_2 & \doteq_i T_2 \mid T_1 \\
 & (T_1 \mid T_2) \mid T_3 & \doteq_i T_1 \mid (T_2 \mid T_3) \\
 & \mathsf{in}\ (A_1 \cup A_2) & \doteq_i \mathsf{in}\ A_1 \mid \mathsf{in}\ A_2 \\
 & \mathsf{out}\ (A_1 \cup A_2) & \doteq_i \mathsf{out}\ A_1 \mid \mathsf{out}\ A_2 \\
 & \mathsf{open}\ (A_1 \cup A_2) & \doteq_i \mathsf{open}\ A_1 \mid \mathsf{open}\ A_2 \\
 & (A_1 \cup A_2)[T] & \doteq_i A_1[T] \mid A_2[T] \\
 & (\vec{a})^{A_1 \cup A_2} \to T & \doteq_i (\vec{a})^{A_1} \to T \mid (\vec{a})^{A_2} \to T \\
 & \langle \vec{T} \rangle^{A_1 \cup A_2} & \doteq_i \langle \vec{T} \rangle^{A_1} \mid \langle \vec{T} \rangle^{A_2} \\
 & a[T_1 \mid T_2] & \doteq_i a[T_1] \mid a[T_2] \\
 & (\nu a).(\nu b).T & \doteq_i (\nu b).(\nu a).T \\
a \notin \mathsf{fn}(T_1) \Rightarrow & (\nu a).(T_1 \mid T) & \doteq_i T_1 \mid (\nu a).T \\
a \neq b \Rightarrow & (\nu a).b[T] & \doteq_i b[(\nu a).T] \\
 & (\nu a).\mathbf{0} & \doteq_i \mathbf{0}
\end{array}
$$

Fig. 6. Type congruence (definition, $i \geq 1$).

Note that a name type is also a message type.

Definition 7. *The result of substituting the message type U for the name a in the type T, written $T[a := U]$ is defined in Fig. 8.* □

It is easy to see that this definition makes sense, in that it does not depend on the term representation chosen for T.

Lemma 6. *$T[a := U]$ is well-defined iff $a \in \mathsf{fan}(T)$ implies U is a name type.* □

Lemma 7. *Assume that $T_1 \doteq T_2$. Then $T_1[a := U]$ is well-defined iff $T_2[a := U]$ is well-defined, in which case $T_1[a := U] \doteq T_2[a := U]$.* □

3.2 Subtyping. There is an ordering $\leq$ on types, with $T_1 \leq T_2$ meaning that T_1 is a more precise shape than T_2.

Definition 8. *We say that $T_1 \leq T_2$ iff $T_1 \leq_i T_2$ holds for all $i \geq 0$, where $T_1 \leq_0 T_2$ always holds and where for $i \geq 1$, $T_1 \leq_i T_2$ is the least relation satisfying the clauses in Fig 9.* □

	T	$\doteq T$	(TypEq Refl)
$T_1 \doteq T_2 \Rightarrow$	T_2	$\doteq T_1$	(TypEq Symm)
$T_1 \doteq T_2$ and $T_2 \doteq T_3 \Rightarrow$	T_1	$\doteq T_3$	(TypEq Trans)
$T_1 \doteq T_2 \Rightarrow$	$(\nu a).T_1$	$\doteq (\nu a).T_2$	(TypEq Res)
$T_1 \doteq T_2 \Rightarrow$	$T_1 \mid T$	$\doteq T_2 \mid T$	(TypEq Par)
$T_1 \doteq T_2 \Rightarrow$	$a[T_1]$	$\doteq a[T_2]$	(TypEq Amb)
$T_1 \doteq T_2 \Rightarrow$	$(\vec{a})^{\lambda} \to T_1$	$\doteq (\vec{a})^{\lambda} \to T_2$	(TypEq Input)
$U_1 \doteq U_2 \Rightarrow$	$\langle \vec{U_1} \rangle^{\lambda}$	$\doteq \langle \vec{U_2} \rangle^{\lambda}$	
	$T \mid \mathbf{0}$	$\doteq T$	(TypEq ZeroPar)
	$T \mid T$	$\doteq T$	(TypEq ParIdem)
	$T_1 \mid T_2$	$\doteq T_2 \mid T_1$	(TypEq ParComm)
	$(T_1 \mid T_2) \mid T_3$	$\doteq T_1 \mid (T_2 \mid T_3)$	(TypEq ParAssoc)
	$\mathsf{in}\ (A_1 \cup A_2)$	$\doteq \mathsf{in}\ A_1 \mid \mathsf{in}\ A_2$	(TypEq InList)
	$\mathsf{out}\ (A_1 \cup A_2)$	$\doteq \mathsf{out}\ A_1 \mid \mathsf{out}\ A_2$	(TypEq OutList)
	$\mathsf{open}\ (A_1 \cup A_2)$	$\doteq \mathsf{open}\ A_1 \mid \mathsf{open}\ A_2$	(TypEq OpenList)
	$(A_1 \cup A_2)[T]$	$\doteq A_1[T] \mid A_2[T]$	(TypEq AmbList)
	$(\vec{a})^{A_1 \cup A_2} \to T$	$\doteq (\vec{a})^{A_1} \to T \mid (\vec{a})^{A_2} \to T$	(TypEq InputList)
	$\langle \vec{T} \rangle^{A_1 \cup A_2}$	$\doteq \langle \vec{T} \rangle^{A_1} \mid \langle \vec{T} \rangle^{A_2}$	(TypEq OutputList)
	$a[T_1 \mid T_2]$	$\doteq a[T_1] \mid a[T_2]$	(TypEq ParAmb)
	$(\nu a).(\nu b).T$	$\doteq (\nu b).(\nu a).T$	(TypEq ResRes)
$a \notin \mathsf{fn}(T_1) \Rightarrow$	$(\nu a).(T_1 \mid T)$	$\doteq T_1 \mid (\nu a).T$	(TypEq ResPar)
$a \neq b \Rightarrow$	$(\nu a).b[T]$	$\doteq b[(\nu a).T]$	(TypEq ResAmb)
	$(\nu a).\mathbf{0}$	$\doteq \mathbf{0}$	(TypEq ZeroRes)

Fig. 7. Type congruence (properties that hold).

Lemma 8. *The relation $T_1 \leq T_2$ enjoys the properties listed in Fig. 10.* □

It turns out that $T_1 \leq T_2$ is not the least relation enjoying the properties of Fig. 10, but fortunately we do not need this.

Lemma 9. *Parallel composition $\mid$ is least upper bound wrt. the ordering $\leq$.* □

Lemma 10. *If $T_1 \leq T_2$ then $\mathsf{fan}(T_1) \subseteq \mathsf{fan}(T_2)$ and $\mathsf{fn}(T_1) \subseteq \mathsf{fn}(T_2)$.* □

Note that if we had not been careful w.r.t. the use of i versus $i-1$ when defining $T_1 \leq_i T_2$, we might have ended up with an inconsistent relation.

Lemma 11. *Assume that $T_1 \leq T_2$. If $T_2[a := U]$ is well-defined, also $T_1[a := U]$ is well-defined, and $T_1[a := U] \leq T_2[a := U]$.* □

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 9. *The predicate "T is closed" is the largest predicate satisfying that if T is closed then the clauses in Fig. 11 hold.* □

$$
\begin{array}{lll}
a[a := U] & = U & \\
b[a := U] & = b & \text{if } a \neq b \\
A[a := U] & = A & \text{if } a \notin A \\
A[a := U] & = (A \setminus \{a\}) \cup \mathsf{names}(U) & \text{if } a \in A,\ U \text{ a name type} \\
\Lambda[a := U] & = \Lambda & \text{if } a \notin \mathsf{fn}(\Lambda) \\
\Lambda[a := U] & = (\Lambda \setminus \{\downarrow a\}) \cup \{ \downarrow b \mid b \in \mathsf{names}(U) \} & \text{if } a \in \mathsf{fn}(\Lambda),\ U \text{ a name type} \\
(\mathsf{in}\ A)[a := U] & = \mathsf{in}\ (A[a := U]) & \\
(\mathsf{out}\ A)[a := U] & = \mathsf{out}\ (A[a := U]) & \\
(\mathsf{open}\ A)[a := U] & = \mathsf{open}\ (A[a := U]) & \\
\mathsf{const}[a := U] & = \mathsf{const} & \\
\mathsf{action}[a := U] & = \mathsf{action} & \\
\mathbf{0}[a := U] & = \mathbf{0} & \\
(T_1 \mid T_2)[a := U] & = (T_1[a := U]) \mid (T_2[a := U]) & \\
((\nu b).T)[a := U] & = (\nu b).(T[a := U]) & \text{if } a \neq b, b \notin \mathsf{fn}(U) \\
(A[T])[a := U] & = (A[a := U])[T[a := U]] & \\
((\vec{a})^{\Lambda} \to T)[a := U] & = (\vec{a})^{\Lambda[a:=U]} \to T[a := U] & \text{if } \{\vec{a}\} \cap (\{a\} \cup \mathsf{fn}(U)) = \emptyset \\
(\langle \vec{U} \rangle^{\Lambda})[a := U] & = \langle \vec{U}[a := U] \rangle^{\Lambda[a:=U]} & \\
X[a := U] & = X &
\end{array}
$$

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

Fig. 8. Type substitution.

$$
\begin{array}{rll}
 & \mathbf{0} & \leq_i T \\
T_1 \doteq T_2 \Rightarrow & T_1 & \leq_i T_2 \\
T_1 \leq_i T_2 \text{ and } T_2 \leq_i T_3 \Rightarrow & T_1 & \leq_i T_3 \\
T_1 \leq_i T_2 \Rightarrow & (\nu a).T_1 & \leq_i (\nu a).T_2 \\
T_1 \leq_i T_2 \Rightarrow & T_1 \mid T & \leq_i T_2 \mid T \\
T_1 \leq_{i-1} T_2 \Rightarrow & a[T_1] & \leq_i a[T_2] \\
T_1 \leq_{i-1} T_2 \Rightarrow & (\vec{a})^{\lambda} \to T_1 & \leq_i (\vec{a})^{\lambda} \to T_2 \\
\vec{U_1} \leq_i \vec{U_2} \Rightarrow & \langle \vec{U_1} \rangle^{\lambda} & \leq_i \langle \vec{U_2} \rangle^{\lambda}
\end{array}
$$

Fig. 9. Subtyping (definition, $i \geq 1$).

This is well-defined, since Fig. 11 does in fact define a motonone operator on sets of types.

We could replace the rule (Clos In) by

$$a[T_1] \mid b[T_2] \leq T \text{ and } (\mathsf{in}\ b) \leq T_1 \Rightarrow b[a[T_1]] \leq T$$

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

$$a[T_1] \leq T \text{ and } (\mathsf{in}\ b) \leq T_1 \Rightarrow b[a[T_1]] \leq T$$

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

$$
\begin{array}{rll}
\mathbf{0} & \leq T & \text{(TypSub Zero)} \\
T_1 \doteq T_2 \Rightarrow T_1 & \leq T_2 & \text{(TypSub Refl)} \\
T_1 \leq T_2 \text{ and } T_2 \leq T_3 \Rightarrow T_1 & \leq T_3 & \text{(TypSub Trans)} \\
T_1 \leq T_2 \Rightarrow (\nu a).T_1 & \leq (\nu a).T_2 & \text{(TypSub Res)} \\
T_1 \leq T_2 \Rightarrow T_1 \mid T & \leq T_2 \mid T & \text{(TypSub Par)} \\
T_1 \leq T_2 \Rightarrow a[T_1] & \leq a[T_2] & \text{(TypSub Amb)} \\
T_1 \leq T_2 \Rightarrow (\vec{a})^{\lambda} \to T_1 & \leq (\vec{a})^{\lambda} \to T_2 & \text{(TypSub Input)} \\
\vec{U_1} \leq \vec{U_2} \Rightarrow \langle \vec{U_1} \rangle^{\lambda} & \leq \langle \vec{U_2} \rangle^{\lambda} & \text{(TypSub Output)}
\end{array}
$$

Fig. 10. Subtyping (properties that hold).

T is closed iff

$$
\left.\begin{array}{r} a[T_1] \mid b[T_2] \leq T \\ (\mathsf{in}\ b) \leq T_1 \end{array}\right\} \Rightarrow b[a[T_1] \mid T_2] \leq T \qquad \text{(Clos In)}
$$

$$
\left.\begin{array}{r} b[a[T_1]] \leq T \\ (\mathsf{out}\ b) \leq T_1 \end{array}\right\} \Rightarrow a[T_1] \leq T \qquad \text{(Clos Out)}
$$

$$
\left.\begin{array}{r} a[T_1] \leq T \\ (\mathsf{open}\ a) \leq T \end{array}\right\} \Rightarrow T_1 \leq T \qquad \text{(Clos Open)}
$$

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

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

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

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

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

$$
a[T_1] \leq T \Rightarrow \exists \text{ closed } T_1' : \left\{\begin{array}{l} T_1 \leq T_1' \\ a[T_1'] \leq T \end{array}\right. \qquad \text{(Clos Amb)}
$$

$$
T \doteq (\nu a).T_1 \Rightarrow T_1 \text{ is closed} \qquad \text{(Clos Res)}
$$

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

Fig. 11. Conditions for closedness.

$$\frac{M : U \quad U \leq U'}{M : U'} \text{ (Exp Subsumpt)} \qquad \frac{P : T \quad T \leq T'}{P : T'} \text{ (Proc Subsumpt)}$$

$$\frac{}{a : a} \text{ (Exp } a\text{)} \qquad \frac{}{c : \mathsf{const}} \text{ (Exp Const)}$$

$$\frac{}{\epsilon : \mathsf{action}} \text{ (Exp } \epsilon\text{)} \qquad \frac{M_1 : U_1 \quad M_2 : U_2}{M_1.M_2 : U_1 \mid U_2 \mid \mathsf{action}} \text{ (Exp Action)}$$

$$\frac{}{\mathsf{in}\ a : \mathsf{in}\ a} \text{ (Exp In)} \qquad \frac{}{\mathsf{out}\ a : \mathsf{out}\ a} \text{ (Exp Out)}$$

$$\frac{}{\mathsf{open}\ a : \mathsf{open}\ a} \text{ (Exp Open)} \qquad \frac{}{\mathbf{0} : \mathbf{0}} \text{ (Proc Zero)}$$

$$\frac{P_1 : T_1 \quad P_2 : T_2}{P_1 \mid P_2 : T_1 \mid T_2} \text{ (Proc Par)} \qquad \frac{P : T}{!P : T} \text{ (Proc Repl)}$$

$$\frac{M : U \mid \mathsf{action} \quad P : T}{M.P : U \mid T} \text{ (Proc Action)}$$

$$\frac{P : T}{(\nu a).P : (\nu a).T} \text{ (Proc Res)} \qquad \frac{P : T}{a[P] : a[T]} \text{ (Proc Amb)}$$

$$\frac{P : T}{(\vec{a})^{\lambda}.P : (\vec{a})^{\lambda} \to T} \text{ (Proc Input)} \qquad \frac{M_1 : U_1 \ldots M_k : U_k \quad P : T}{\langle \vec{M} \rangle^{\lambda}.P : \langle \vec{U} \rangle^{\lambda} \mid T} \text{ (Proc Output)}$$

Fig. 12. Our type system.

4 The Shape Type System

In Fig. 12 we define two typing judgements $M : U$ and $P : T$. Most rules are very straightforward; the rule (Proc Repl) reflects that we do not count multiplicities. Thanks to the presence of action in the rules (Exp ϵ) and (Exp Action) (the destruction rule is (Proc Action)), we have the following result which is crucial for our later soundness results.

Lemma 12. *Assume that $M : U$ with U a name type. Then M is a name.* □

Lemma 13. *Assume that $P : T$, that $M : U$, and that $T[a := U]$ is well-defined. Then also $P[a := M]$ is well-defined, and $P[a := M] : T[a := U]$.* □

Theorem 1 (Subject reduction). *If $P \longrightarrow Q$ and $P : T$ with T closed, then $Q : T$.* □

Theorem 2 (Safety). *If $P : T$ with T closed, then execution of P will never give rise to an ill-defined substitution.* □

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

Theorem 3 (Principality, naive). *Given a process P, there exists a type T_P such that*

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

□

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

Conjecture 1 (Principality, hoped for). Given a process P that can be assigned a *closed* type. (Note that not all processes can be assigned a closed type.) Then there exists a type T_P such that

– $P : T_P$;
– T_P is closed;
– if $P : T$ with T closed 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 (Principality, limited). *Assume that we are working in a restriction of our type system which forbids ν and type recursion.*

Given a process P that can be assigned a closed *type. Then there exists a type T_P such that*

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

□

To prove this theorem, we need the following result:

Lemma 14. *Given a non-empty set $\{T_i \mid i \in I\}$ of finite, ν-free types. Then one can construct a (finite and ν-free) type T which is the greatest lower bound of $\{T_i \mid i \in I\}$.*

Moreover, if all T_i are closed also T 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 type:

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

To close this 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 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 Boxed Ambient calculus. However, at some point a type system handling open flexibly would be desirable for new applications in modeling intracellular biological processes.[3]
- 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 $\mathsf{in}\ b$ 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.

5 Embedding Standard Ambient Types

It can be shown that the type system of Cardelli and Gordon [9] can be embedded into our system. 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 [9], by assigning the following types to the (free and bound) names:

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

[3] E.g., see `http://www.wisdom.weizmann.ac.il/~aviv/index_main.html`.

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

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

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.

We expect that a similar embedding can be done for the mobility type system of [7]. The translation would then give P a shape type which is more refined than the one above, but still much coarser than the best shape type for P which is:

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

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 [18], 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 [17], 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 [14] 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. For instance, it seems possible to embed the types into the logic of [6].
– 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 naive application of the rules in Fig. 11 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 query language based on the ambient logic. In *ESOP 2001, Genova*, vol. 2028 of *LNCS*. Springer-Verlag, 2001.

[7] 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.

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

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

[10] P. Cousot, R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe, M. Wirsing, eds., *PLILP'92*, vol. 631 of *LNCS*. Springer-Verlag, 1992.

[11] C. Fournet, G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, 1996. ACM.

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

[13] J. C. Godskesen, T. Hildebrandt, V. Sassone. A calculus of mobile resources. In *CONCUR'02*, vol. 2421 of *LNCS*. Springer-Verlag, 2002.

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

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

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

[17] 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.

[18] 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.

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

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