Shape Types for Ambients with Communication Dependencies

Torben Amtoft and Joe Wells

Heriot-Watt University
Ambient Calculi

The Ambient Calculus is a process calculus designed by Cardelli & Gordon, and later extended and modified by many others, to model these notions:

- **Location**: All processes are located in *ambients* which can be nested, forming a tree.
The Ambient Calculus is a process calculus designed by Cardelli & Gordon, and later extended and modified by many others, to model these notions:

- **Location**: All processes are located in *ambients* which can be nested, forming a tree.

- **Mobility**: Ambients can move, making the tree dynamic.
Ambient Calculi

The Ambient Calculus is a process calculus designed by Cardelli & Gordon, and later extended and modified by many others, to model these notions:

- **Location**: All processes are located in *ambients* which can be nested, forming a tree.
- **Mobility**: Ambients can move, making the tree dynamic.
- **Communication**: Processes that are “*close*” to each other can exchange values.
Example Rewrite Sequence

\[ q_1 \xrightarrow{p_1} (x)^* . x [\text{out } p_1 . 0] \langle p_1 , r \rangle^* . 0 \quad \xrightarrow{q} \text{in } q_1 . 0 \mid (p , v)^\uparrow . \text{in } p . \langle v \rangle^\uparrow . 0 \]
Example Rewrite Sequence

$q_1 \quad \frac{p_1 \quad (x)^* \cdot x[\text{out } p_1 \cdot 0]}{\langle p_1, r \rangle^* \cdot 0}$

$q \quad \text{in } q_1 \cdot 0 \mid (p, v)^\uparrow \cdot \text{in } p \cdot \langle v \rangle^\uparrow \cdot 0$

$q_1 \quad \frac{p_1 \quad (x)^* \cdot x[\text{out } p_1 \cdot 0]}{\langle p_1, r \rangle^* \cdot 0}$

$q \quad (p, v)^\uparrow \cdot \text{in } p \cdot \langle v \rangle^\uparrow \cdot 0$
Example Rewrite Sequence

\[ q_1 \]

\[ p_1 \]

\[
(x)^* . x \text{[out } p_1.0 \text{]} \quad \langle p_1, r \rangle^*.0
\]

\[
q \quad \text{in } q_1.0 \mid (p, v)^\uparrow . \text{in } p . \langle v \rangle^\uparrow .0
\]

\[ q_1 \]

\[ p_1 \]

\[
(x)^* . x \text{[out } p_1.0 \text{]} \quad \langle p_1, r \rangle^*.0
\]

\[
q \quad (p, v)^\uparrow . \text{in } p . \langle v \rangle^\uparrow .0
\]

\[ q_1 \]

\[ p_1 \]

\[
(x)^* . x \text{[out } p_1.0 \text{]}
\]

\[
q \quad \text{in } p_1 . \langle r \rangle^\uparrow .0
\]
Example Rewrite Sequence

\[
q_1 \quad p_1 \quad q
\]
\[
\begin{array}{c}
(x)^* . x [\text{out } p_1.0] \\
\end{array}
\]
\[
\begin{array}{c}
\langle p_1, r \rangle^*.0 \\
\end{array}
\]
\[
\begin{array}{c}
in q_1.0 \mid (p, v)^+.\text{in } p.\langle v \rangle^+.0 \\
\end{array}
\]
\[
q
\]

\[
q_1 \quad p_1 \quad q
\]
\[
\begin{array}{c}
(x)^* . x [\text{out } p_1.0] \\
\end{array}
\]
\[
\begin{array}{c}
\langle p_1, r \rangle^*.0 \\
\end{array}
\]
\[
\begin{array}{c}
(p, v)^+.\text{in } p.\langle v \rangle^+.0 \\
\end{array}
\]

\[
q
\]

\[
q_1 \quad p_1 \quad q
\]
\[
\begin{array}{c}
(x)^* . x [\text{out } p_1.0] \\
\end{array}
\]
\[
\begin{array}{c}
in q_1.\langle r \rangle^+.0 \\
\end{array}
\]

\[
q
\]

\[
q_1 \quad p_1 \quad q
\]
\[
\begin{array}{c}
(x)^* . x [\text{out } p_1.0] \\
\end{array}
\]
\[
\begin{array}{c}
\langle r \rangle^+.0 \\
\end{array}
\]

\[
q
\]
Example Rewrite Sequence

\[ q_1 \quad \begin{array}{c} p_1 \\ \langle x \rangle^*.x \text{out } p_{1.0} \end{array} \]
\[ q \quad \begin{array}{c} \langle p_1, r \rangle^*.0 \\ \text{in } q_1.0 \mid (p, v)^\uparrow.\text{in } p.(v)^\uparrow.0 \end{array} \]

\[ q_1 \quad \begin{array}{c} p_1 \\ \langle x \rangle^*.x \text{out } p_{1.0} \end{array} \]
\[ q \quad \begin{array}{c} \langle p_1, r \rangle^*.0 \\ (p, v)^\uparrow.\text{in } p.(v)^\uparrow.0 \end{array} \]

\[ q_1 \quad \begin{array}{c} p_1 \\ \langle x \rangle^*.x \text{out } p_{1.0} \end{array} \]
\[ q \quad \begin{array}{c} \text{in } p_1.(r)^\uparrow.0 \end{array} \]

\[ q_1 \quad \begin{array}{c} p_1 \\ \langle x \rangle^*.x \text{out } p_{1.0} \end{array} \]
\[ q \quad \begin{array}{c} \langle r \rangle^\uparrow.0 \end{array} \]
\[ q_1 \quad \begin{array}{c} p_1 \\ \text{out } p_{1.0} \end{array} \]

Useful Logics, Types, Rewriting, and their Automation

Shape Types for Ambients with Communication Dependencies – p.3/17
Example Rewrite Sequence

$q_1 \rightarrow p_1 \langle x \cdot \cdot x \text{out } p_1.0 \rangle \langle p_1, r \rangle^\cdot .0 \rightarrow q \text{ in } q_1.0 \mid (p, v)^\uparrow \text{in } p.\langle v \rangle^\uparrow .0$

$q_1 \rightarrow p_1 \langle x \cdot \cdot x \text{out } p_1.0 \rangle \langle p_1, r \rangle^\cdot .0 \rightarrow q \rightarrow (p, v)^\uparrow \text{in } p.\langle v \rangle^\uparrow .0$

$q_1 \rightarrow p_1 \langle x \cdot \cdot x \text{out } p_1.0 \rangle \rightarrow q \rightarrow \text{in } p_1.\langle r \rangle^\uparrow .0$

$q_1 \rightarrow p_1 \langle x \cdot \cdot x \text{out } p_1.0 \rangle \rightarrow q \rightarrow \langle r \rangle^\uparrow .0$
Example Rewrite Sequence

\[ q_1 \]

\[
\begin{array}{c}
p_1 \\
(x) . x [\text{out } p_1.0] \\
\langle p_1, r \rangle^*.0 \\
\end{array}
\]

\[ q \]

\[
\begin{array}{c}
\text{in } q_1.0 | (p, v)^\uparrow \text{in } p.\langle v \rangle^\uparrow.0 \\
\end{array}
\]

\[ q_1[p_1[(x) . x [\text{out } p_1.0]] | \langle p_1, r \rangle^*.0] | q[q_1.0 | (p, v)^\uparrow \text{in } p.\langle v \rangle^\uparrow.0] \\
\]

\[ q_1 \]

\[
\begin{array}{c}
p_1 \\
(x) . x [\text{out } p_1.0] \\
\langle p_1, r \rangle^*.0 \\
\end{array}
\]

\[ q \]

\[
\begin{array}{c}
(p, v)^\uparrow \text{in } p.\langle v \rangle^\uparrow.0 \\
\end{array}
\]

\[ q_1 \]

\[
\begin{array}{c}
p_1 \\
(x) . x [\text{out } p_1.0] \\
\end{array}
\]

\[ q \]

\[
\begin{array}{c}
\text{in } p_1.\langle r \rangle^\uparrow.0 \\
\end{array}
\]

\[ q_1 \]

\[
\begin{array}{c}
p_1 \\
(x) . x [\text{out } p_1.0] \\
\langle r \rangle^\uparrow.0 \\
\end{array}
\]

\[ q \]

\[
\begin{array}{c}
\text{out } p_1.0 \\
\end{array}
\]

\[ q_1 \]

\[
\begin{array}{c}
p_1 \\
\end{array}
\]

\[ q \]

\[
\begin{array}{c}
r \\
\end{array}
\]

\[ q \]

\[
\begin{array}{c}
\text{out } p_1.0 \\
\end{array}
\]
The Language Formalism

We use the extension of Boxed Ambients with \texttt{open}.

\[
\begin{align*}
\lambda \in \text{Locs} & : = \star \mid \uparrow \mid \downarrow a \\
M \in \text{Exp} & : = a \mid c \mid \text{in } a \mid \text{out } a \mid \text{open } a \mid \epsilon \mid M_1.M_2 \\
P, Q, R \in \text{Proc} & : = 0 \mid P_1 \mid P_2 \mid !P \mid (\nu a).P \\
& \quad \mid M.P \mid a[P] \mid (\tilde{a})^\lambda.P \mid \langle \overrightarrow{M} \rangle^\lambda.P
\end{align*}
\]
The Language Formalism

We use the extension of Boxed Ambients with open.

\[ \lambda \in \text{Locs} ::= \star \mid \uparrow \mid \downarrow a \]

\[ M \in \text{Exp} ::= a \mid c \mid \text{in } a \mid \text{out } a \mid \text{open } a \mid \epsilon \mid M_1.M_2 \]

\[ P, Q, R \in \text{Proc} ::= 0 \mid P_1 \mid P_2 \mid !P \mid (\nu a).P \]

\[ \mid M.P \mid a[P] \mid (\tilde{a})^\lambda.P \mid \langle \tilde{M} \rangle^\lambda.P \]

The semantics are defined by rewrite rules such as:

\[ b[(\tilde{a})^{\uparrow}.P \mid R] \mid \langle \tilde{M} \rangle^\ast.Q \rightarrow b[P[\tilde{a} := \tilde{M}] \mid R] \mid Q \]
The Language Formalism

We use the extension of Boxed Ambients with open.

\[
\begin{align*}
\lambda & \in \text{Locs} \quad ::= \quad * \mid \uparrow \mid \downarrow a \\
M & \in \text{Exp} \quad ::= \quad a \mid c \mid \text{in} a \mid \text{out} a \mid \text{open} a \mid \epsilon \mid M_1.M_2 \\
P, Q, R & \in \text{Proc} \quad ::= \quad 0 \mid P_1 \mid P_2 \mid !P \mid (\nu a).P \\
& \quad \mid M.P \mid a[P] \mid (\bar{a})^\lambda.P \mid \langle \bar{M} \rangle^\lambda.P
\end{align*}
\]

The semantics are defined by rewrite rules such as:

\[
b[(\bar{a})^\uparrow.P \mid R] \mid \langle \bar{M} \rangle^* .Q \longrightarrow b[P[\bar{a} := \bar{M}] \mid R] \mid Q
\]

Substitution may not always be well-defined:

\[
\begin{align*}
\text{(in } a)[a := \text{out } b] & \quad (b[a.0])[a := \text{out } c] & \quad (a[\text{in } b])[a := \epsilon]
\end{align*}
\]
The Language Formalism

We use the extension of Boxed Ambients with open.

\[ \lambda \in \text{Locs} := \ast \mid \uparrow \mid \downarrow a \]
\[ M \in \text{Exp} := a \mid c \mid \text{in } a \mid \text{out } a \mid \text{open } a \mid \epsilon \mid M_1 . M_2 \]
\[ P, Q, R \in \text{Proc} := 0 \mid P_1 \mid P_2 \mid !P \mid (\nu a).P \]
\[ \mid M . P \mid a[P] \mid (\bar{a})^\lambda . P \mid \langle M \rangle^\lambda . P \]

The semantics are defined by rewrite rules such as:

\[ b[(\bar{a})^{\uparrow} . P \mid R] \mid \langle M \rangle^{\ast} . Q \longrightarrow b[P[\bar{a} := M] \mid R] \mid Q \]

Substitution may not always be well-defined:

\[ (\text{in } a)[a := \text{out } b] \quad (b[a.0])[a := \text{out } c] \quad (a[\text{in } b])[a := \epsilon] \]

A type system should ensure at least well-definedness.
Previous Type Systems

Remember our example:

\[ q_1[\langle p_1, r \rangle^* . 0 \mid p_1[\langle x \rangle^* . x[\text{out } p_1 . 0]]] \]
\[ \mid q[\text{in } q_1 . 0 \mid (p, v)^{\uparrow} . \text{in } p. \langle v \rangle^{\uparrow} . 0] \]
Previous Type Systems

Remember our example:

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

This can be typed by the system for Boxed Ambients (in the spirit of Cardelli & Gordon’s original) because in each ambient the “topic of conversation” is well-defined (for each arity and direction):

\[
\text{type of } v, x : T_1 \quad = \quad \text{Amb}[\text{shh}, \text{shh}]
\]
\[
\text{type of } p, p1 : T_2 \quad = \quad \text{Amb}[T_1, \text{shh}]
\]
\[
\text{type of } q : T \quad = \quad \text{Amb}[\text{shh}, T_1 \cup (T_2 \times T_1)]
\]
Previous Type Systems

Remember our example:

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

This can be typed by the system for Boxed Ambients (in the spirit of Cardelli & Gordon’s original) because in each ambient the “topic of conversation” is well-defined (for each arity and direction):

\[
\text{type of } v, x : T_1 = \text{Amb[shh, shh]}
\]
\[
\text{type of } p, p1 : T_2 = \text{Amb}[T_1, \text{shh}]
\]
\[
\text{type of } q : T = \text{Amb[shh, } T_1 \cup (T_2 \times T_1)\text{]}
\]

Note: It is unclear what such type assumptions really mean.
Example Needing Poly-morphism/variance

We extend the previous example process to have *two* possible execution paths:

\[
q_1[\langle p_1, r \rangle^* .0 | \ p_1[(x)^*.x[\text{out} \ p_1.0]]] \quad (x \ \text{must be a name})
\]

\[
| \ q_2[\langle p_2, \text{out} \ p_2 \rangle^* .0 | \ p_2[(x)^*.r[x.0]]] \quad (x \ \text{must be a capability})
\]

\[
| \ q[\text{in} \ q_1.0 | \text{in} \ q_2.0 | (p, v)^+.\text{in} \ p.\langle v \rangle^+.0]
\]
Example Needing Poly-morphism/variance

We extend the previous example process to have two possible execution paths:

\[ q_1[p_1, r]^* . 0 | p_1[(x)^* . x[\text{out } p_1.0]] ] \quad (x \text{ must be a name}) \]

\[ | \quad q_2[p_2, \text{out } p_2]^* . 0 | p_2[(x)^* . r[x.0]] ] \quad (x \text{ must be a capability}) \]

\[ | \quad q[\text{in } q_1.0 | \text{in } q_2.0 | (p, v)^\uparrow . \text{in } p. \langle v \rangle^\uparrow . 0] \]

So the type of \( v \) has to be both an ambient name and a capability. None of the existing type systems allow this.
Example Needing Poly-morphism/variance

We extend the previous example process to have two possible execution paths:

\[
q_1[\langle p_1, r \rangle^*.0 | p_1[(x)^*.x[\text{out } p_1.0]]] \quad (x \text{ must be a name})
\]

\[
| q_2[\langle p_2, \text{out } p_2 \rangle^*.0 | p_2[(x)^*.r[x.0]]] \quad (x \text{ must be a capability})
\]

\[
| q[\text{in } q_1.0 | \text{in } q_2.0 | (p, v)\uparrow.\text{in } p.\langle v \rangle\uparrow.0]
\]

So the type of \(v\) has to be both an ambient name and a capability. None of the existing type systems allow this.

Key observation: the topic of conversation within \(q\) depends on whether \(q\) is inside \(q_1\) or inside \(q_2\).
Example Needing Poly-morphism/variance

We extend the previous example process to have two possible execution paths:

\[
q_1[\langle p_1, r \rangle^* .0 | p_1[(x)^* .x[\text{out } p_1 .0]]] \quad (x \text{ must be a name})
\]
\[
| q_2[\langle p_2, \text{out } p_2 \rangle^* .0 | p_2[(x)^* .r[x .0]]] \quad (x \text{ must be a capability})
\]
\[
| q[\text{in } q_1 .0 | \text{in } q_2 .0 | (p, v)^\uparrow .\text{in } p.\langle v \rangle^\uparrow .0]
\]

So the type of \( v \) has to be both an ambient name and a capability. None of the existing type systems allow this.

Key observation: the topic of conversation within \( q \) depends on whether \( q \) is inside \( q_1 \) or inside \( q_2 \).

To overcome some of the weaknesses of previous type systems for ambient calculi, we will now present a new type system.
The New Type System (1)

The types represent upper bounds on the possible ambient nesting tree into which a process can evolve, e.g.:

\[
\left( a[\text{in } b.0] \mid b[0] \right) : \left( a[\text{in } b] \mid b[a[\text{in } b]] \right)
\]
The New Type System (1)

- The types represent upper bounds on the possible ambient nesting tree into which a process can evolve, e.g.:

\[ (a \text{in } b.0 \mid b[0]) : (a \text{in } b \mid b[a \text{in } b]) \]

- Types indicate the possible positions of capabilities, inputs, and outputs.
The New Type System (1)

The types represent upper bounds on the possible ambient nesting tree into which a process can evolve, e.g.:

\[(a \text{in } b.0 \parallel b[0]) : (a \text{in } b \parallel b[a\text{in } b])\]

- Types indicate the possible positions of capabilities, inputs, and outputs.
- The types say nothing about the number of copies of a feature at a location.
The New Type System (2)

There are singleton types of ambient names and explicit dependencies on communication, e.g.:

\[
(x)^*.x[0] \mid \langle a \rangle^*.0 \quad : \quad ((x)^* \rightarrow x[0]) \mid \langle a \rangle^* \mid a[0]
\]
The New Type System (2)

- There are singleton types of ambient names and explicit dependencies on communication, e.g.:

\[
\left( (x)^* . x[0] \mid \langle a \rangle^* . 0 \right) : \left( ((x)^* \rightarrow x[0]) \mid \langle a \rangle^* \mid a[0] \right)
\]

- Sequential composition is replaced by parallel composition, except for inputs, e.g.:

\[
\left( p[\text{in } q . \text{in } r . 0] \mid r[0] \right) : \left( p[\text{in } q \mid \text{in } r] \mid r[p[\text{in } q \mid \text{in } r]] \right)
\]
The New Type System (2)

There are singleton types of ambient names and explicit dependencies on communication, e.g.:

\[
((x)^* \cdot a[0] \mid (a)^* \cdot 0) : ((x)^* \rightarrow x[0]) \mid (a)^* \mid a[0]
\]

Sequential composition is replaced by parallel composition, except for inputs, e.g.:

\[
(p[\text{in } q, \text{in } r] \cdot 0 \mid r[0]) : (p[\text{in } q, \text{in } r, r[\text{in } q, \text{in } r]]
\]

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

\[a[T_1] \mid a[T_2] \vdash a[T_1 \mid T_2]\]
The New Type System (3)

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

\[
\left( !a[!\text{in } a.0] \right) : \left( \text{letrec } X = a[\text{in } a \mid X] \text{ in } X \right)
\]
Types can be infinitely deep trees, e.g.:

\[
(!a[!\text{in } a.0]) : \left(\text{letrec } X = a[\text{in } a \mid X \text{ in } X]\right)
\]

We only consider types that can be given a finite term representation. Due to binders, their precise characterization is non-trivial (Glew, ESOP ’02).
The New Type System (3)

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

\[
\left( \!a[\! \text{in } a.0] \right) : \left( \text{letrec } X = a[\text{in } a \mid X \text{ in } X] \right)
\]

We only consider types that can be given a finite term representation. Due to binders, their precise characterization is non-trivial (Glew, ESOP ’02).

For our convenience, there is only one sort of types which is used for both messages (a.k.a. expressions) and processes.
Typing for Example Program

\[ q_1[\langle p_1, r \rangle^* . 0 \mid p_1[(x)^* . x[out \ p_1.0] ] ] \quad (x \text{ must be a name}) \]

\[ q_2[\langle p_2, out \ p_2 \rangle^* . 0 \mid p_2[(x)^* . r[x . 0] ] ] \quad (x \text{ must be a capability}) \]

\[ q[in \ q_1.0 \mid in \ q_2.0 \mid (p, v)^\uparrow.in \ p. \langle v \rangle^\uparrow .0] \]
Subtyping and Closedness

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

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

Parallel composition ("|") is least upper bound w.r.t. that ordering.
Subtyping and Closedness

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

Parallel composition ("\|") is least upper bound w.r.t. that ordering.

We demand that types are closed under certain rules that simulate rewriting, such as:

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

$$({\bar{a}})^* \rightarrow T' \leq T \text{ and } \langle\bar{T}\rangle^* \leq T \quad \Rightarrow \quad T'[\bar{a} := \bar{T}] \leq T$$

(implying LHS well-defined)

To compute the closure, approximations are needed.
Type System

\[
\frac{P_1 : T_1 \quad P_2 : T_2}{P_1 \mid P_2 : T_1 \mid T_2}
\]

(very intuitive, as most rules)
Type System

\[ \frac{P_1 : T_1 \quad P_2 : T_2}{P_1 \parallel P_2 : T_1 \parallel T_2} \]

(very intuitive, as most rules)

\[ \frac{P : T}{!P : T} \]

(as no multiplicities)
Type System

\[
P_1 : T_1 \quad P_2 : T_2
\]
\[
P_1 \parallel P_2 : T_1 \parallel T_2
\]

\[
P : T
\]
\[
!P : T
\]

\[
M_1 : T_1 \quad M_2 : T_2
\]
\[
M_1.M_2 : T_1 \parallel T_2 \parallel \text{action}
\]

(very intuitive, as most rules)

(as no multiplicities)

(need to record it’s not a name)
Type System

\[
\begin{align*}
    & P_1 : T_1 \quad P_2 : T_2 \\
    \Rightarrow & \quad P_1 \parallel P_2 : T_1 \parallel T_2 \\
    & P : T \\
    \Rightarrow & \quad \lnot P : T \\
    & M_1 : T_1 \quad M_2 : T_2 \\
    \Rightarrow & \quad M_1.M_2 : T_1 \parallel T_2 \text{ action} \\
    & M : T_0 \quad P : T \\
    \Rightarrow & \quad M.P : T_0 \parallel T
\end{align*}
\]

(very intuitive, as most rules)

(as no multiplicities)

(need to record it’s not a name)
Type System

\[
\begin{align*}
P_1 : T_1 & \quad P_2 : T_2 \\
\frac{\text{} & \quad P_1 \parallel P_2 : T_1 \parallel T_2}{P : T} \\
\frac{P : T}{!P : T} \\
\frac{M_1 : T_1 \quad M_2 : T_2}{M_1.M_2 : T_1 \parallel T_2 \text{ action}} \\
\frac{M : T_0 \quad P : T}{M.P : T_0 \parallel T}
\end{align*}
\]

(very intuitive, as most rules)

(as no multiplicities)

(need to record it’s not a name)

Subject reduction: If \( P \rightarrow Q \) and \( P : T \) (thus \( T \) closed), then \( Q : T \).
Type System

\[
P_1 : T_1 \quad P_2 : T_2
\]
\[
P_1 \mid P_2 : T_1 \mid T_2
\]
\[
P : T
\]
\[
!P : T
\]
\[
M_1 : T_1 \quad M_2 : T_2
\]
\[
M_1.M_2 : T_1 \mid T_2 \mid \text{action}
\]
\[
M : T_0 \quad P : T
\]
\[
M.P : T_0 \mid T
\]

(very intuitive, as most rules)

(as no multiplicities)

(need to record it’s not a name)

**Subject reduction:** If \( P \rightarrow Q \) and \( P : T \) (thus \( T \) closed), then \( Q : T \).

**Safety:** If \( P : T \) (thus \( T \) closed), then execution of \( P \) will never give rise to an ill-defined substitution.
Consider this term:

$$a[\text{in } b.0] \parallel b[\text{open } a.0]$$

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

$$a[\text{in } b] \parallel b[\text{open } a]$$
Consider this term:  
\[ a[\text{in } b.0] \mid b[\text{open } a.0] \]

Ignoring the closedness requirement, we could give it this type:  
\[ a[\text{in } b] \mid b[\text{open } a] \]

To close this type, observe that \(a\) can go into \(b\):  
\[ a[\text{in } b] \mid b[a[\text{in } b] \mid \text{open } a] \]
Consider this term:

\[ a[\text{in } b.0] \mid b[\text{open } a.0] \]

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

\[ a[\text{in } b] \mid b[\text{open } a] \]

To close this type, observe that \( a \) can go into \( b \):

\[ a[\text{in } b] \mid b[a[\text{in } b] \mid \text{open } a] \]

Then \( a \) can be opened:

\[ a[\text{in } b] \mid b[a[\text{in } b] \mid \text{open } a \mid \text{in } b] \]
Anomaly with open

Consider this term:

\[ a[\text{in } b.0] \parallel b[\text{open } a.0] \]

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

\[ a[\text{in } b] \parallel b[\text{open } a] \]

To close this type, observe that \( a \) can go into \( b \):

\[ a[\text{in } b] \parallel b[a[\text{in } b] \parallel \text{open } a] \]

Then \( a \) can be opened:

\[ a[\text{in } b] \parallel b[a[\text{in } b] \parallel \text{open } a \parallel \text{in } b] \]

Now, one copy of \( b \) can go into another:

\[ \ldots \parallel b[\ldots \parallel \text{in } b \parallel b[\ldots \parallel \text{in } b]] \]
Consider this term:

\[ a[\text{in } b.0] | b[\text{open } a.0] \]

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

\[ a[\text{in } b] | b[\text{open } a] \]

To close this type, observe that \( a \) can go into \( b \):

\[ a[\text{in } b] | b[a[\text{in } b] | \text{open } a] \]

Then \( a \) can be opened:

\[ a[\text{in } b] | b[a[\text{in } b] | \text{open } a | \text{in } b] \]

Now, one copy of \( b \) can go into another:

\[ \ldots | b[\ldots | \text{in } b | b[\ldots | \text{in } b]] \]

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

\[ a[\text{in } b] | \text{let rec } X = b[a[\text{in } b] | \text{open } a | \text{in } b | X] | \text{in } X \]
What to do about the anomaly?

Ignore it. This is probably okay, but the types would look ugly.
What to do about the anomaly?

- **Ignore it.** This is probably okay, but the types would look ugly.
- **Avoid using** open. As is the recent trend, for instance in Boxed Ambients.

Note: the ability to dissolve an ambient will be crucial for new applications in modeling intracellular biological processes.
What to do about the anomaly?

- **Ignore it.** This is probably okay, but the types would look ugly.

- **Avoid using** `open`. As is the recent trend, for instance in Boxed Ambients.

  Note: the ability to dissolve an ambient will be crucial for new applications in modeling intracellular biological processes.

- **Multiplicities.** Counting is not enough because the types “confuse the past with the future”, e.g., the count $\omega$ must be $\omega$ to make this type closed:

  $$(\text{open } a)^1 \mid a[(\text{in } b)^1]^1 \mid (\text{in } b)\omega$$
What to do about the anomaly?

- **Ignore it.** This is probably okay, but the types would look ugly.

- **Avoid using** `open`. As is the recent trend, for instance in Boxed Ambients.

  Note: the ability to dissolve an ambient will be crucial for new applications in modeling intracellular biological processes.

- **Multiplicities.** Counting is not enough because the types “confuse the past with the future”, e.g., the count $\tau$ must be $\omega$ to make this type closed:

  $$ (\text{open } a)^1 \parallel a[(\text{in } b)^1]^1 \parallel (\text{in } b)^{\tau} $$

- **Union types.** Together with multiplicities, union types could work theoretically, but they are not feasible because each point in the possible future state space would likely become a separate type.
At least for $\nu$-free programs, we can translate a typing.

$$a[\text{open } b.(y)^*.\text{in } y.0 \mid b[c]^*.0] \mid c[0]$$
At least for $\nu$-free programs, we can translate a typing.

\[ a[\text{open } b.(y)^* \cdot \text{in } y.0 \mid b[\langle c \rangle^* .0]] \mid c[0] \]

They assign $T_1 = \text{Amb}[\text{shh}]$ to $c$ and $y$

They assign $T_2 = \text{Amb}[T_1]$ to $a$ and $b$. 
Embedding Cardelli & Gordon (POPL '99)

At least for $\nu$-free programs, we can translate a typing:

$$a[\text{open } b.(y)^*.\text{in } y.0 \mid b[\langle c \rangle^*.0]] \mid c[0]$$

They assign $T_1 = \text{Amb}[\text{shh}]$ to $c$ and $y$

They assign $T_2 = \text{Amb}[T_1]$ to $a$ and $b$.

A mechanical translation converts the above into this type:

```latex
\text{letrec} \quad X_C = \text{in } \{a, b, c, y\} \mid \text{out } \{a, b, c, y\} \mid \text{action} \\
X_A = a[X_1] \mid b[X_1] \mid c[X_0] \\
X_0 = X_A \mid X_C \mid \text{open } \{c, y\} \\
X_1 = X_A \mid X_C \mid \text{open } \{a, b\} \mid \langle c \rangle^* \mid \langle y \rangle^* \mid (y)^* \rightarrow X_1 \\
\text{in } X_0
```
Embedding Cardelli & Gordon (POPL '99)

At least for \( \nu \)-free programs, we can translate a typing:

\[
a[\text{open } b.(y)^* \cdot \text{in } y.0 \mid b[\langle c \rangle^* \cdot 0]] \mid c[0]
\]

They assign \( T_1 = \text{Amb[shh]} \) to \( c \) and \( y \)

They assign \( T_2 = \text{Amb}[T_1] \) to \( a \) and \( b \).

A mechanical translation converts the above into this type:

\[
\begin{align*}
\text{letrec } & \quad X_C = \text{in } \{a, b, c, y\} \mid \text{out } \{a, b, c, y\} \mid \text{action} \\
& \quad X_A = a[X_1] \mid b[X_1] \mid c[X_0] \\
& \quad X_0 = X_A \mid X_C \mid \text{open } \{c, y\} \\
& \quad X_1 = X_A \mid X_C \mid \text{open } \{a, b\} \mid \langle c \rangle^* \mid \langle y \rangle^* \mid (y)^* \rightarrow X_1 \\
& \quad \text{in } X_0
\end{align*}
\]

This can probably be extended to Mobility Types (Cardelli & Ghelli & Gordon).
Other Kinds of Poly-morphic/variant Analysis

- **Shape grammars** (Nielson & Nielson, POPL ’00)
  Returns a set of grammars such that at any step, the current process can be described by one of these grammars.
  Very precise, but potentially also very expensive.
Other Kinds of Poly-morphic/variant Analysis

- **Shape grammars** (Nielson & Nielson, POPL ’00)
  Returns a set of grammars such that at any step, the current process can be described by one of these grammars. Very precise, but potentially also very expensive.

- **Kleene Analysis** (Nielson & Nielson & Sagiv, ESOP ’00)
  Using 3-valued logic, estimates the possible shapes. Trade-offs w.r.t. precision vs. costs.
Other Kinds of Poly-morphic/variant Analysis

- Shape grammars (Nielson & Nielson, POPL ’00)
  Returns a set of grammars such that at any step, the current process can be described by one of these grammars. Very precise, but potentially also very expensive.

- Kleene Analysis (Nielson & Nielson & Sagiv, ESOP ’00)
  Using 3-valued logic, estimates the possible shapes. Trade-offs w.r.t. precision vs. costs.

- Abstract Interpretation (Levi & Maffeis, SAS ’01)
  Keeps track of the context “one level up”. Quite precise, and yet “only” polynomial ($n^7$).
Other Kinds of Poly-morphic/variant Analysis

- **Shape grammars** (Nielson & Nielson, POPL ’00)
  Returns a set of grammars such that at any step, the current process can be described by one of these grammars. Very precise, but potentially also very expensive.

- **Kleene Analysis** (Nielson & Nielson & Sagiv, ESOP ’00)
  Using 3-valued logic, estimates the possible shapes. Trade-offs w.r.t. precision vs. costs.

- **Abstract Interpretation** (Levi & Maffeis, SAS ’01)
  Keeps track of the context “one level up”. Quite precise, and yet “only” polynomial ($n^7$).

None of the above-listed work handles communication, so none can show our example is safe.
We presented a type system for a variant ambient calculus that:

- Is poly-morphic/variant in that an ambient is analyzed differently for different interactions it enters into.

Future work includes:

- Writing a terminating algorithm for computing closure.
- Investigating the relationship to other systems. For instance, it seems possible to embed the types into the logic of Cardelli & Ghelli (ESOP ’01).
- Evaluating practical usefulness and feasibility.
Conclusion

We presented a type system for a variant ambient calculus that:

- Is poly-morphic/variant in that an ambient is analyzed differently for different interactions it enters into.
- Has dependent typing where the analysis tracks which values are communicated and reacts accordingly.
Conclusion

We presented a type system for a variant ambient calculus that:

- Is poly-morphic/variant in that an ambient is analyzed differently for different interactions it enters into.

- Has dependent typing where the analysis tracks which values are communicated and reacts accordingly.

Future work includes:

- Writing a terminating algorithm for computing closure.
Conclusion

We presented a type system for a variant ambient calculus that:

- Is poly-morphic/variant in that an ambient is analyzed differently for different interactions it enters into.
- Has dependent typing where the analysis tracks which values are communicated and reacts accordingly.

Future work includes:

- Writing a terminating algorithm for computing closure.
- Investigating the relationship to other systems. For instance, it seems possible to embed the types into the logic of Cardelli & Ghelli (ESOP ’01).
Conclusion

We presented a type system for a variant ambient calculus that:

- Is poly-morphic/variant in that an ambient is analyzed differently for different interactions it enters into.
- Has dependent typing where the analysis tracks which values are communicated and reacts accordingly.

Future work includes:

- Writing a terminating algorithm for computing closure.
- Investigating the relationship to other systems. For instance, it seems possible to embed the types into the logic of Cardelli & Ghelli (ESOP ’01).
- Evaluating practical usefulness and feasibility.