POLYA: TRUE TYPE POLYMORPHISM FOR MOBILE AMBIENTS

Torben Amtoft[†]
Kansas State University

Henning Makholm
Heriot-Watt University

J. B. Wells
Heriot-Watt University

Abstract Previous type systems for mobility calculi (the original Mobile Ambients, its variants and descendants, e.g., Boxed Ambients and Safe Ambients, and other related systems) offer little support for generic mobile agents. Previous systems either do not handle communication at all or globally assign fixed communication types to ambient names that do not change as an ambient moves around or interacts with other ambients. This makes it hard to type examples such as a messenger ambient that uses communication primitives to collect a message of non-predetermined type and deliver it to a non-predetermined destination.

In contrast, we present our new type system PolyA. Instead of assigning communication types to ambient names, PolyA assigns a type to each process P that gives upper bounds on (1) the possible ambient nesting shapes of any process P' to which P can evolve, (2) the values that may be communicated at each location, and (3) the capabilities that can be used at each location. Because PolyA can type generic mobile agents, we believe PolyA is the first type system for a mobility calculus that provides type polymorphism comparable in power to polymorphic type systems for the **λ-calculus.** PolyA is easily extended to ambient calculus variants. A restriction of PolyA has principal typings.

1 Introduction

Whereas the **π-calculus** [15] is probably the most widely known calculus for communicating processes, the ambient calculus [6] has recently become important, because it adds reasoning about locations and mobility. In the ambient calculus, pro-

* Partially supported by EC FP5 grant IST-2001-33477, EPSRC grant GR/R41545/01, NSF grants 9806745 (EIA), 9988529 (CCR), and 0113193 (ITR), and Sun Microsystems equipment grant EDUD-7826-990410-US.

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

cesses are located in *ambients,* locations which can be nested, forming a tree. Ambients can move, making the tree dynamic. Furthermore, only processes that are "close" to each other can exchange values.

1.1 The problem with ambient calculus type systems

Consider this process:

$$\mathtt{m[in\ s.0 \mid open\ t.(p,v).p.\langle v\rangle.0] \mid s[t[in\ m.\langle in\ r, d\rangle.0] \mid r[open\ m.(v).out\ v.0]]}$$

The example ambient named m is perhaps the simplest kind of *generic mobile agent,* namely a *messenger.* That is, m first goes somewhere looking for messages to deliver, then m collects a destination and a payload, and then m goes to that destination and delivers that payload.

Nearly all type systems for ambient calculi follow the example of the seminal system of Cardelli and Gordon [7] and assign to each ambient name *a* a description of the communication that can happen within ambients named *a.* Unfortunately, type systems based on this principle are inflexible about generic functionality. Consider the example process extended to have *two* possible execution paths, in that m can enter either of two senders:

$$\begin{array}{ll} & \mathtt{m[in\ s.0 \mid open\ t.(p,v).p.\langle v\rangle.0]} \\ \mid & \mathtt{s[t[in\ m.\langle in\ r, d\rangle.0] \mid r[open\ m.(v).out\ v.0]]} \quad \textbf{(v \textit{must be a name})} \\ \mid & \mathtt{s[t[in\ m.\langle in\ q, out\ d\rangle.0] \mid q[open\ m.(v).v.0]]} \quad \textbf{(v \textit{must be a capability})} \end{array}$$

Here, the messenger m 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.

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. However, this encoding is awkward and also loses the ability to predict whether the correct type of data is being delivered to each location, avoiding stuck states.

In solving this problem, a key observation is that the possible communication within m depends on which of the s's the ambient m is found inside.

1.2 Our solution – overview

To overcome the weaknesses of previous type systems for generic functionality, we present a new type system, PolyA. Types 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, e.g., for the types of [9].

Our type system's basic concept is the *shape predicate.* The actual definition is somewhat involved, partly due to the need of handling communication, so let us introduce the concept gently with a toy system where the only capability is "in":

$$\textbf{Toy shape predicates: } \sigma \ ::= \ 0 \quad (\sigma \mid \sigma) \quad a[\sigma] \quad \text{in } a$$

A shape predicate's meaning is a set of terms, given by this matching relation:

$$\frac{\vdash P : \sigma}{\vdash a[P] : (\cdots \mid a[\sigma] \mid \cdots)} \qquad \frac{\vdash P : \sigma \quad \vdash Q : \sigma}{\vdash P \mid Q : \sigma}$$

$$\frac{}{\vdash \mathsf{in}\ a.0 : (\cdots \mid \mathsf{in}\ a \mid \cdots)} \qquad \frac{}{\vdash 0 : \sigma}$$

With these rules we can derive the judgement $\vdash P_0 : \sigma_0$, where

$$P_0 = \mathtt{a[in\ b.0 \mid in\ c.0] \mid b[d[in\ a.0]] \mid c[e[in\ a.0]]}$$
$$\sigma_0 = \mathtt{a[in\ b \mid in\ c] \mid b[d[in\ a]] \mid c[e[in\ a]]}$$

But we can also derive, say,

$$\vdash \mathtt{a[in\ b.0] \mid a[in\ c.0]} : \sigma_0$$

— the matching rules do not care that the `b` and `c` on the top level are missing, nor that the **a[in b | in c]** part of the shape predicate is used twice.

PolyA *types* are shape predicates such that the set of terms matching a type is closed under reduction. The shape predicate σ_0 above is not a type, because

$$P_0 \hookrightarrow P_1 = \mathtt{b[a[in\ c.0] \mid d[in\ a.0]] \mid c[\cdots]}$$

yet $\not\vdash P_1 : \sigma_0$. One type that P_0 does have is

$$\sigma_1 = \mathtt{a[in\ b \mid in\ c] \mid b[a[in\ b \mid in\ c \mid d[in\ a]] \mid d[in\ a]]} \\ \mathtt{\mid c[a[in\ b \mid in\ c \mid e[in\ a]] \mid e[in\ a]]}$$

The `a[...]` predicate inside `b` still allows the in `b`. This must be so because shape predicates do not care about the number of identical items (unlike what is the case in [19]), so one of the terms matched by σ_1 is **a[in b.0 | in b.0] | b[0]**, which reduces to `b[a[inb]]`.

A more subtle point about σ_1 is that it *disallows* having an `e` inside an `a` inside a `b`, or a `d` inside an a inside a `c`. This example therefore illustrates the most basic kind of polymorphism possible: The same initial `a` ambient can evolve differently in different possible futures, and the type system can prove that those different futures do not interfere with each other.

PolyA lets any supertype (i.e., a type that is matched by a larger set of terms) be used as a *polymorphic variant* if it appears in the right place of the overall typing. The overall typing contains all of the polymorphic variants that will ever be needed for each ambient in the particular context it is being typed in.

Some readers might think that this does not *look* like type polymorphism, because the various types for `a` are not substitution instances of a *parameterised* type. However, how one technically expresses the relation between the type for some generic code and the types for its concrete uses is not essential to the concept of genericity or polymorphism. What is important is that the type system supports reasoning about distinct uses of the same generic code. We achieve what Cardelli and Wegner [8] called "the purest form of polymorphism: the same object or function can be used uniformly in different type context without changes, coercions or any kind of run-time tests or special encodings of representations".

PolyA can optionally track the sequencing of actions, a possibility pioneered by Amtoft et al. [1,2]. For example, **a[in b.in c.0] | b[c[0]] | c[open a.0]** has a PolyA type proving that a will never be opened.

PolyA can assign the following type to the example containing the generic messenger and two clients:

```
letrec Xm8 = in s.0 | open t.(p,v).p.⟨{v}⟩.0
in m[Xm8]
 | s[letrec Xm7 = (Xm8) | (Xt3) | ⟨{d}⟩.0 | (p,v).p.⟨{v}⟩.0
                  | in r.⟨{d}⟩.0 | m[Xm7] | r[Xr1] | t[Xt3]
          Xr1= (Xm7) | (v).out v.0 | out d.0 | open m.(v).out v.0
          Xt3= ⟨<in r>,{d}⟩.0 | in m.⟨<in r>,{d}⟩.0
     in m[Xm7] | r[Xr1] | t[in m.⟨<in r>,{d}⟩.0] end]
 | s[letrec Xm3 = (Xm8) | (Xt1) | ⟨<out d>⟩.0 | (p,v).p.⟨{v}⟩.0
                  | in q.⟨<out d>⟩.0 | m[Xm3] | q[Xq1] | t[Xt1]
          Xq1= (Xm3) | (v).v.0 | out d.0 | open m.(v).v.0
          Xt1= ⟨<in q>,<out d>⟩.0 | in m.⟨<in q>,<out d>⟩.0
     in m[Xm3] | q[Xq1] | t[in m.⟨<in q>,<out d>⟩.0] end]
end
```

This type proves that the example process has only well defined behaviour, something which no previous type system for ambients can do. The type may appear complex compared to the term it types. This is partly because we constructed it with the help of a type inference algorithm [14] which strives to create a very precise (and thus information-rich) type. It is possible to construct visually smaller but less precise types that also prove well defined behaviour for the messenger example.

1.3 Other related work

Although not type-based, several papers have explored 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. None of these handle communication, however, so none can prove the safety of our example polymorphic messenger. With shape grammars [17], a set of grammars is returned 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. In Kleene analysis [16], a 3-valued logic is used to estimate the possible shapes. The framework allows for trade-offs w.r.t. precision versus costs. The abstract interpretation system of [11] keeps track of the context "one level up". This is sufficient to achieve a quite precise analysis, yet is "only" polynomial (n^7).

Polymorphic type systems already exist for the **π-calculus** [20, 18], but do not generalise easily to the spatial nature of our messenger example.

1.4 Summary of contributions (conclusion)

- We present PolyA, the first type system for the ambient calculus that is flexible enough to type generic mobile agents.
- We explain how PolyA types can be used not just to check basic type safety but also to give precise answers to various questions about process behaviour of interest for other reasons, e.g., security.

- We prove subject reduction (Thm. 16) and the decidability of type checking (Prop. 6) for PolyA.
- We prove principal typings (Thm. 23) for a useful restriction of PolyA.
- We illustrate how to extend PolyA to support the cross-ambient communication of Boxed Ambients [4], the co-capabilities of Safe Ambients [12], and the process (not ambient) mobility capability of M³ [10].

The proofs of most propositions and theorems have been omitted here for space reasons. They can be found in an extended online version of this paper [3].

In other work [14] we have developed a type inference algorithm for a useful restriction of PolyA. Space limitations prevent including a further description here.

Acknowledgements The design of PolyA benefited from helpful discussions with Mario Coppo, Mariangiola Dezani, and Elio Giovannetti.

2 The ambient calculus

For space reasons, we present the system for a calculus without name restriction. In [3] we present a straightforward way to handle name restriction. In later work it may be possible to combine PolyA with more advanced treatments of name restriction, such as the "abstract names" of Lhoussaine and Sassone [13].

Fig. 1 defines the syntax and semantics of our base calculus. Whenever it has been defined that some (meta)variable letter, say "x", ranges over a given set of objects, the notation $\boxed{x}$ shall mean that set of objects.

The syntactic category of *prefixes* is not in traditional ambient calculus formulations. Our calculus treats ambient boundaries as capabilities; "amb a" is the capability that creates an ambient named a when executed. In our formulation, an ambient with contents P is written "amb $a.P$". The traditional notation "a[P]" is syntactic sugar for amb $a.P$; we use this whenever convenient. The capability amb a can in principle be passed in a message. We allow this more because it is syntactically convenient than because we expect processes to actually do it. Our main results do not fully support programs that *use* this possibility.

The special capability **"•"** is not supposed to be found in the initial term. It signifies a substitution result that would otherwise be syntactically invalid. For example, the term ⟨in c⟩ | (b).in a.open b.0 reduces to in a.•.0 instead of the (hypothetical) "in a.open (in c).0". Traditional ambient calculus accounts usually leave such a communication result undefined, implicitly understanding that the system would crash either at the communication time or when the ill-formed capability executes after the in a capability has fired.

The symbol • does not have any reduction rules associated with it. As far as our theory is concerned it just sits there. Likewise, there are no reduction rules for placeholder capabilities of the form "a". A PolyA type conservatively approximates *whether* and *where* one of these capabilities may occur, but the type system user must decide whether or not to consider it an error if this happens.

CONVENTION 1 *A term P is* ***well formed*** *iff its free names are distinct from the names bound by any* **"$(\vec{a})$"** *within the term and it does not contain any nested bindings of the same name. We consider only well formed terms.*

Syntax:

$$
\begin{array}{lrcl}
\text{Names:} & a,b & ::= & \mathsf{a} \mid \mathsf{b} \mid \mathsf{c} \mid \cdots \\
\text{Opcodes:} & O & ::= & \mathsf{in} \mid \mathsf{out} \mid \mathsf{open} \mid \mathsf{amb} \\
\text{Capabilities:} & C & ::= & a \mid Oa \mid \bullet \\
\text{Messages:} & M,N & ::= & C \mid M.N \mid \varepsilon \\
\text{Prefixes:} & p & ::= & M \mid \langle \vec{M} \rangle \mid (\vec{a}) \\
\text{Processes:} & P,Q,R & ::= & p.P \mid\ !P \mid (P \mid Q) \mid 0
\end{array}
$$

See main text for further syntactic restrictions (scoping).

Process equivalence:

$$
\frac{}{P \mid Q \equiv Q \mid P} \qquad \frac{}{P \mid (Q \mid R) \equiv (P \mid Q) \mid R} \qquad \frac{}{0 \mid P \equiv P} \qquad \frac{}{!P \equiv P \mid !P}
$$

$$
\frac{}{!0 \equiv 0} \qquad \frac{}{(M.N).P \equiv M.(N.P)} \qquad \frac{}{\varepsilon.P \equiv P} \qquad \frac{}{P \equiv P}
$$

$$
\frac{P \equiv Q}{p.P \equiv p.Q} \qquad \frac{P \equiv Q}{P \mid R \equiv Q \mid R} \qquad \frac{P \equiv Q}{!P \equiv\ !Q} \qquad \frac{Q \equiv P}{P \equiv Q} \qquad \frac{P \equiv Q \quad Q \equiv R}{P \equiv R}
$$

Substitution:

A **term substitution** $\mathcal{S}$ is a (total) function from names to messages such that $\mathcal{S}(a) \neq a$ for only finitely many a's. We often notate it $\mathcal{S} = [a_1 \mapsto M_1, \ldots, a_k \mapsto M_k]$, understanding implicitly that $\mathcal{S}(a) = a$ when a is not one of the a_i's. Shorter notations are $[a_i \mapsto M_i]_{1 \le i \le k}$ or $[a \mapsto M_a]_{a \in A}$.

For messages:

$$
\begin{array}{rlrl}
\mathcal{S}(M.N) &= (\mathcal{S}M).(\mathcal{S}N) & \mathcal{S}\varepsilon &= \varepsilon \\
\mathcal{S}a &= \mathcal{S}(a) & \mathcal{S}\bullet &= \bullet \\
\mathcal{S}(Oa) &= \begin{cases} O\mathcal{S}(a) & \text{if } \mathcal{S}(a) \text{ is a name} \\ \bullet & \text{otherwise} \end{cases} & &
\end{array}
$$

For other prefixes:

$$
\begin{array}{rl}
\mathcal{S}\langle M_1, \ldots, M_k \rangle &= \langle \mathcal{S}M_1, \ldots, \mathcal{S}M_k \rangle \\
\mathcal{S}(a_1, \ldots, a_k) &= \begin{cases} (a_1, \ldots, a_k) & \text{if } \mathcal{S}(a_i) = a_i \text{ for all } i \\ \bullet & \text{otherwise} \end{cases}
\end{array}
$$

For terms:

$$
\begin{array}{rlrl}
\mathcal{S}(p.P) &= (\mathcal{S}p).(\mathcal{S}P) & \mathcal{S}(!P) &= !(\mathcal{S}P) \\
\mathcal{S}(P \mid Q) &= (\mathcal{S}P) \mid (\mathcal{S}Q) & \mathcal{S}0 &= 0
\end{array}
$$

Reduction rules:

$$
\frac{}{a[\mathsf{in}\ b.P \mid Q] \mid b[R] \hookrightarrow b[a[P \mid Q] \mid R]}
$$

$$
\frac{}{b[a[\mathsf{out}\ b.P \mid Q] \mid R] \hookrightarrow a[P \mid Q] \mid b[R]} \qquad \frac{}{a[P] \mid \mathsf{open}\ a.Q \hookrightarrow P \mid Q}
$$

$$
\frac{}{\langle M_1, \ldots, M_n \rangle.P \mid (a_1, \ldots, a_n).Q \hookrightarrow P \mid [a_i \mapsto M_i]_{1 \le i \le n} Q}
$$

$$
\frac{P \hookrightarrow Q}{a[P] \hookrightarrow a[Q]} \qquad \frac{P \hookrightarrow Q}{P \mid R \hookrightarrow Q \mid R} \qquad \frac{P \equiv P' \quad P' \hookrightarrow Q' \quad Q' \equiv Q}{P \hookrightarrow Q}
$$

Figure 1. Syntax and semantics of the ambient calculus

Conv. 1 does not limit expressiveness. Any program (term) in a more conventional ambient calculus formulation that allows **α-conversion** has a well formed **α-variant** which can be used in our type system.

The convention ensures that our reduction rules will never perform a substitution where there is a risk of name capture by $(\vec{a})$ bindings. Reductions preserve well-formedness, because it is syntactically impossible for a substitution to inject a $(\vec{a})$ within the body of another $(\vec{a})$. (This is in contrast to the **λ-calculus,** where substitutions routinely insert **λ-abstractions** into other abstractions). Because of this, we do not need to recognise **α-equivalence** for $(\vec{a}).P$. This is a significant technical simplification, because for many purposes we can treat $(\vec{a})$ as any other action, without needing special machinery for **α-equivalence** of the bound names.

Fig. 1 contains no provisions for avoiding name capture in $\mathcal{S}(\vec{a})$ — this is handled by Convention 1. The • possibility for $\mathcal{S}(\vec{a})$ is never supposed to be used; substitutions leading to it will not arise by our rules.

3 Shape predicates

The following pseudo-grammar defines the (abstract) syntax of our type system:

$$
\begin{array}{lll}
\textbf{Message types:} & \mu ::= \{C_1, C_2, \cdots, C_k\}* & (C_i\text{'s all different}, k \geq 1) \\
& \quad | \quad \langle C_1.C_2.\cdots.C_k \rangle & (C_i\text{'s all different}, k \geq 0) \\
& \quad | \quad \{a\} & \\
\textbf{Prefix types:} & \pi ::= C \ (\vec{a}) \ \langle \vec{\mu} \rangle & \\
\textbf{Shape predicates:} & \sigma ::= (\pi_1.\sigma_1 \mid \cdots \mid \pi_k.\sigma_k) & (k \geq 1) \\
& \quad | \quad 0 &
\end{array}
$$

DEFINITION 2 (MATCHING OF SHAPE PREDICATES) *These rules define the relations* $\vdash M : \mu$, $\vdash p : \pi$, *and* $\vdash P : \sigma$*:*

$$\frac{M \notin \boxed{a} \qquad M.0 \equiv C_1'.\cdots.C_n'.0 \qquad \{C_1', \ldots, C_n'\} \subseteq \{C_1, \ldots, C_k\}}{\vdash M : \{C_1, \ldots, C_k\}*} \ \textsf{KleeneStar}$$

$$\frac{M \notin \boxed{a} \qquad M.0 \equiv C_1.\cdots.C_k.0}{\vdash M : \langle C_1.\cdots.C_k \rangle} \ \textsf{Sequenced} \qquad \frac{}{\vdash a : \{a\}} \ \textsf{Name}$$

$$\frac{}{\vdash C : C} \ \textsf{Cap} \qquad \frac{}{\vdash (\vec{a}) : (\vec{a})} \ \textsf{Recv} \qquad \frac{\vdash M_1 : \mu_1 \quad \cdots \quad \vdash M_k : \mu_k}{\vdash \langle M_1, \ldots, M_k \rangle : \langle \mu_1, \ldots, \mu_k \rangle} \ \textsf{Send}$$

$$\frac{\vdash p : \pi \qquad \vdash P : \sigma}{\vdash p.P : (\cdots \mid \pi.\sigma \mid \cdots)} \ \textsf{Pfx} \qquad \frac{\vdash M.(N.P) : \sigma}{\vdash (M.N).P : \sigma} \ \textsf{Seq} \qquad \frac{\vdash P : \sigma}{\vdash \varepsilon.P : \sigma} \ \textsf{Nop}$$

$$\frac{\vdash P : \sigma \qquad \vdash Q : \sigma}{\vdash P \mid Q : \sigma} \ \textsf{Par} \qquad \frac{}{\vdash 0 : \sigma} \ \textsf{Null} \qquad \frac{\vdash P : \sigma}{\vdash !P : \sigma} \ \textsf{Bang}$$

The side conditions $M \notin \boxed{a}$ and $M.0 \equiv C_1.\cdots.C_k.0$ on rules KleeneStar and Sequenced amount to specifying that these two forms of message types are matched modulo associativity of "." and neutrality of **"ε"** — with the exception that messages that are raw names (i.e., *"a"* as opposed to **"*a*.ε"** or "in *a*") are handled specially. They are matched only by the message type $\{a\}$.

THEOREM 3 *If* $P \equiv Q$ *then* $\vdash P : \sigma \Leftrightarrow \vdash Q : \sigma$ *for all* σ.

DEFINITION 4 *The* ***meaning*** *of a shape predicate (message type, prefix type) is the set of terms (messages, prefixes) that match it:*

$$[\![\mu]\!] = \{M \mid \vdash M : \mu\} \qquad [\![\pi]\!] = \{p \mid \vdash p : \pi\} \qquad [\![\sigma]\!] = \{P \mid \vdash P : \sigma\}$$

DEFINITION 5 *Define the following* ***containment*** *relations:*

$$\mu \leq \mu' \iff [\![\mu]\!] \subseteq [\![\mu']\!] \qquad \pi \leq \pi' \iff [\![\pi]\!] \subseteq [\![\pi']\!] \qquad \sigma \leq \sigma' \iff [\![\sigma]\!] \subseteq [\![\sigma']\!]$$

Each of the three containment relations is apreorder (transitive and reflexive). Containment of shape predicates is not antisymmetric, however. For example, the shape predicates amb `a.amb b.0` and amb `a.amb b.0` | amb `a.0` have the same meaning, but it would be technically inconvenient (and not give any real benefit) to insist on equating shape predicates with equal meanings.

3.1 Recursive shape predicates

Our strategy in analysing a term is to look for a shape predicate describing all of its possible computational futures. Because many terms can create arbitrarily deep nestings of ambients (e.g., `!a[!in a.0]`), the finite trees we have used for shape predicates so far are not up to the task[1]. We need infinite shape predicates. We should, however, restrict ourselves to infinite shape predicates with finite *representations* — in other words, regular trees.

There are several regular tree representations that we could have used. We believe it is technically most convenient (and intuitive) to view regular trees as *graphs.* Therefore, we retroactively replace the abstract syntax for shape predicates with:

Node identifiers:	X, Y, Z	$::= \mathbf{X1} \quad \mathbf{X2} \quad \mathbf{X3} \quad \cdots$
Edges:	e	$::= X \xrightarrow{\pi} Y$
Shape graphs:	G	$\in \mathcal{P}_{\text{fin}}(\boxed{e})$
Shape predicates:	σ	$::= \langle X \mid G \rangle$

A shape predicate is now a shape graph together with a pointer to a distinguished *root* node. The version of the Pfx rule that works with this notation is

$$\frac{\vdash p : \pi \qquad X \xrightarrow{\pi} Y \in G \qquad \vdash P : \langle Y \mid G \rangle}{\vdash p.P : \langle X \mid G \rangle}\ \text{Pfx}$$

Thm. 3 is still true with this formulation, because it was proven by induction on term equivalence rather than shape-predicate structure.

This graph-based formulation is the basis for our formal development. However, even though graphs are an intuitive way of *thinking* about regularly infinite shape predicates, they are less convenient for *writing down* shape predicates. Figure 2 defines a more tree-like textual notation for shape graphs for use in examples.

[1] This happens even for terminating terms such as `b[in a.0] | a[open b.0]`, which shape predicates cannot distinguish from `!b[!in a.0] | !a[open b.0]`. Thus, nearly every nontrivial use of open will need recursive σ's. As already observed by [5], open often complicates analysis significantly.

This is the syntax of **shape expressions**:

Shape expressions: $V ::= X \mid U \mid \text{letrec } X_1 = U_1; \ldots; X_n = U_n \text{ in } X_i$
Shape summands: $U ::= 0 \mid (U \mid U) \mid \pi \mid (\pi_1 \mid \cdots \mid \pi_n).V \mid (X)$

As additional syntactic sugar, letrec $\cdots$ in U stands for letrec $\cdots; X = U$ in X where X is fresh. $\pi.V$ stands for $(\pi).V$, and $a[V]$ stands for $(\text{amb } a).V$.

To convert a shape expression to a graph-shaped shape predicate, first replace each U of the form (X) with the right-hand side of the innermost in-scope letrec binding for X. It is an error if no such binding exist, of if the unfolding does not terminate. (V's of the form X are not touched at this stage). Then α-rename the entire shape expression such that no X is bound by two different letrec's, and apply the function $(\cdot)^*$ defined by:

a) V^* is a shape predicate:

$$X^* = \langle X \mid \varnothing \rangle \qquad U^* = \langle X \mid U^*_X \rangle \text{ where } X \text{ is fresh}$$
$$(\text{letrec } X_1 = U_1; \ldots; X_n = U_n \text{ in } X_i)^* = \langle X_i \mid U_1{}^*_{X_1} \cup \cdots \cup U_n{}^*_{X_n} \rangle$$

b) U^*_X is a shape graph:

$$0^*_X = \varnothing \qquad (U_1 \mid U_2)^*_X = (U_1)^*_X \cup (U_2)^*_X$$
$$\pi^*_X = \{X \xrightarrow{\pi} X\} \qquad ((\pi_1 \mid \cdots \mid \pi_n).V)^*_X = \{X \xrightarrow{\pi_i} X' \mid 1 \le i \le n\} \cup G$$
$$\text{where } \langle X' \mid G \rangle = V^*$$

Note that the parentheses in $U ::= (X)$ are important; they distinguish between "$\pi.X$", which stands for an edge going to node X itself, and "$\pi.(X)$", which stands for an edge to a fresh node that happens to behave like X. This can influence whether the shape graph is "modest" (or "discrete"); see Sect. 4.3.

Figure 2. Shape expressions: a tree-like notation for recursive shape predicates

In general, defining some property for shape graphs implicitly defines it for shape predicates: The shape predicate $\langle X \mid G \rangle$ has the property iff G has.

PROPOSITION 6 *The relations of Defn. 2 are effectively (and efficiently) decidable when shape predicates are given as graphs.*

DEFINITION 7 *Two shape graphs* G_1 *and* G_2 *are* ***equivalent****, written* $G_1 \approx G_2$*, iff* $[\![\langle X \mid G_1 \rangle]\!] = [\![\langle X \mid G_2 \rangle]\!]$ *for all X.*

3.2 Effective characterisation of containment

DEFINITION 8 *Let R be a relation between shape predicates. R is a* ***shape simulation*** *iff* $\langle X \mid G \rangle \; R \; \langle X' \mid G' \rangle$ *and* $X \xrightarrow{\pi} Y \in G$ *imply that there is* $\pi' \ge \pi$ *and* Y' *such that* $X' \xrightarrow{\pi'} Y' \in G'$ *and* $\langle Y \mid G \rangle \; R \; \langle Y' \mid G' \rangle$.

THEOREM 9 *Shape containment* $\le$ *is the largest shape simulation; it is the union of all shape simulations.*

Thus, to prove that $\sigma \le \sigma'$ it is sufficient to find a shape simulation R such that $\sigma \; R \; \sigma'$. This strategy leads directly to:

PROPOSITION 10 *The relation* $\langle X \mid G\rangle \leq \langle X' \mid G'\rangle$ *can be decided effectively (actually, in polynomial time).*

It is worth noticing that shape simulations treat $(\vec{a})$ just like any other prefix type. Thus $\leq$ treats the "result" type covariantly (like [22]), whereas the input position in PolyA is a list of names and thus essentially invariant.

3.3 Type substitutions

DEFINITION 11 *A* ***type substitution*** $\mathcal{T}$ *is a function from names to message types such that* $\mathcal{T}(a) \neq \{a\}$ *for only finitely many a's. Like term substitutions, type substitutions may be written as* $[a_1 \mapsto \mu_1, \ldots, a_k \mapsto \mu_k]$ *or* $[a \mapsto \mu_a]_{a \in A}$.

A type substitution can be applied to capabilities, message types, shape graphs, and shape predicates as follows:

Type substitution for capabilities: $\mathcal{T}C$ is a message type, not a capability.

$$\mathcal{T}a = \mathcal{T}(a) \qquad \mathcal{T}(Oa) = \begin{cases} \langle Ob\rangle & \text{if } \mathcal{T}(a) = \{b\} \\ \langle\bullet\rangle & \text{otherwise} \end{cases} \qquad \mathcal{T}\bullet = \langle\bullet\rangle$$

Substitution for message types: $\mathcal{T}\mu$ is a message type given by:

To compute $\mathcal{T}\{C_1, \ldots, C_k\}*$, let $\mu_i = \mathcal{T}C_i$ for $1 \leq i \leq k$. If $\mu_i = \langle\rangle$ for all i, then the result is also $\langle\rangle$. Otherwise, the result is $\{C'_1, \ldots, C'_n\}*$, where the C'_js are all capabilities that occur in any of the μ_i's, with duplicates removed (and in some canonical order).

To compute $\mathcal{T}\langle C_1 \cdots C_k\rangle$, let $\mu_i = \mathcal{T}C_i$ for $1 \leq i \leq k$. If any μ_i has the form $\{\cdots\}*$, or if any C appears in more than one μ_i, then the result is the same as the result of $\mathcal{T}\{C_1, \ldots, C_k\}*$. Otherwise, each μ_i has the form $\langle \cdots \rangle$. Concatenate all of the capability lists (in the order of the i's) and return <the concatenated list>.

Finally, $\mathcal{T}\{a\}$ is simply $\mathcal{T}(a)$.

Substitution for shape graphs: $\mathcal{T}G$ is a shape graph. To construct $\mathcal{T}G$, first construct an intermediate graph G_ε which can contain special null edges written $X \xrightarrow{\varepsilon} Y$. G_ε contains contributions from each edge $Y_1 \xrightarrow{\pi} Y_2 \in G$:

1 When $\pi = a$ and $\mathcal{T}(a) = \{C_1, \ldots, C_k\}*$, choose a fresh node Z, and add to G_ε the edges: $Y_1 \xrightarrow{\varepsilon} Z \xrightarrow{C_1} Z \xrightarrow{C_2} \cdots \xrightarrow{C_k} Z \xrightarrow{\varepsilon} Y_2$

2 When $\pi = a$ and $\mathcal{T}(a) = \langle C_1 \cdots C_k\rangle$, choose fresh nodes Z_0 through Z_k, and add to G_ε the edges $Y_1 \xrightarrow{\varepsilon} Z_0 \xrightarrow{C_1} Z_1 \xrightarrow{C_2} \cdots Z_k \xrightarrow{\varepsilon} Y_2$

3 When $\pi = a$ and $\mathcal{T}(a) = \{b\}$, add to G_ε the edge $Y_1 \xrightarrow{b} Y_2$.
When $\pi = Oa$, $\mathcal{T}(Oa)$ will always have the form $\langle C'\rangle$. Add to G_ε the edge $Y_1 \xrightarrow{C'} Y_2$.

4 When $\pi = (a_1, \ldots, a_k)$, check that $\mathcal{T}a_i = \{a_i\}$ for all i, and then add the edge $Y_1 \xrightarrow{\pi} Y_2$ to G_ε. Otherwise, add $Y_1 \xrightarrow{\bullet} Y_2$.

5 When $\pi = \langle \mu_1, \ldots, \mu_k\rangle$, add to G_ε the edge $Y_1 \xrightarrow{\langle \mathcal{T}\mu_1, \ldots, \mathcal{T}\mu_l\rangle} Y_2$.

Now set $\mathcal{T}G = \{X_k \xrightarrow{\pi} Y \mid (X_k \xrightarrow{\varepsilon} X_{k-1} \xrightarrow{\varepsilon} \cdots \xrightarrow{\varepsilon} X_0 \xrightarrow{\pi} Y) \in G_\varepsilon, k \geq 0\}$.

Substitution for shape predicates: $\mathcal{T}\sigma$ is a shape predicate given by:

$$\mathcal{T}\langle X \mid G\rangle = \langle X \mid \mathcal{T}G\rangle$$

THEOREM 12 *Assume that* $\vdash P : \sigma$ *and* $\vdash \mathcal{S}(a) : \mathcal{T}(a)$ *for all a. Then* $\vdash \mathcal{S}P : \mathcal{T}\sigma$.

4 Shape predicates as types

4.1 Closed shape predicates

DEFINITION 13 *The shape predicate* σ *is* ***semantically closed*** *iff its meaning is closed under reduction, i.e., if* $\vdash P : \sigma$ *and* $P \hookrightarrow Q$ *imply* $\vdash Q : \sigma$.

This definition is intuitively appealing, but it is not immediately clear how to decide it. However, we have local rules that imply semantic closure:

DEFINITION 14 *The shape graph G is* ***locally closed*** *at* X_0 *iff*

1. $\{(X_0 \xrightarrow{\text{amb } a} X), (X \xrightarrow{\text{in } b} Y), (X_0 \xrightarrow{\text{amb } b} Z)\} \subseteq G$
   $\Rightarrow \exists X' : Z \xrightarrow{\text{amb } a} X' \in G \wedge \langle X \mid G\rangle \leq \langle X' \mid G\rangle \wedge \langle Y \mid G\rangle \leq \langle X' \mid G\rangle$,
2. $\{(X_0 \xrightarrow{\text{amb } a} X), (X \xrightarrow{\text{amb } b} Y), (Y \xrightarrow{\text{out } a} Z)\} \subseteq G$
   $\Rightarrow \exists Y' : X_0 \xrightarrow{\text{amb } b} Y' \in G \wedge \langle Y \mid G\rangle \leq \langle Y' \mid G\rangle \wedge \langle Z \mid G\rangle \leq \langle Y' \mid G\rangle$,
3. $\{(X_0 \xrightarrow{\text{amb } a} X), (X_0 \xrightarrow{\text{open } a} Y)\} \subseteq G$
   $\Rightarrow \langle X \mid G\rangle \leq \langle X_0 \mid G\rangle \wedge \langle Y \mid G\rangle \leq \langle X_0 \mid G\rangle$, *and*
4. $\{(X_0 \xrightarrow{\langle \mu_1, \ldots, \mu_k \rangle} Y), (X_0 \xrightarrow{(a_1, \ldots, a_k)} Z)\} \subseteq G$
   $\Rightarrow \langle Y \mid G\rangle \leq \langle X_0 \mid G\rangle \wedge [a_i \mapsto \mu_i]_{1 \leq i \leq k} \langle Z \mid G\rangle \leq \langle X_0 \mid G\rangle$.

DEFINITION 15 *Let* $\sigma = \langle X \mid G\rangle$ *be a shape predicate. The* ***active nodes*** *in* σ, *written* $\mathsf{active}(\sigma)$, *is the least set of node names such that*

$$\mathsf{active}(\sigma) = \{X\} \cup \{Z \mid \exists Y \in \mathsf{active}(\sigma) : \exists a : Y \xrightarrow{\text{amb } a} Z \in G\}.$$

The predicate σ *is* ***syntactically closed*** *iff G is locally closed at every* $X \in \mathsf{active}(\sigma)$.

THEOREM 16 *Every syntactically closed shape predicate is also semantically closed.*

4.2 Types

DEFINITION 17 *A* ***type*** τ *is a syntactically closed shape predicate. Given a type* τ, *the term P has type* τ *iff* $\vdash P : \tau$.

This notion of types has the basic properties expected of any type system: It enjoys subject reduction (Thm. 16), it can be effectively decided whether a given term has a given type (Prop. 6), and types can be distinguished from non-types (using Prop. 10).

Given an algorithm to compute precise types, (such as the one we present in [14]), one can approximate various properties of a term's computational behaviour:

- If P has the type $\sigma = \langle X \mid G\rangle$ and G contains no edge $Y \xrightarrow{\bullet} Z$ with $Y \in \mathsf{active}(\sigma)$, then P will never *execute* the result of a bad substitution such as $[a \mapsto M.N](\text{in } a)$.
- If P has the type $\langle X \mid G\rangle$ and G contains no edge $Y \xrightarrow{\bullet} Z$, then executing P will never create such a malformed substitution result.

- Any **security policy** can be checked if it can be stated as a condition on configurations that must not arise. For example, the policy "no ambient a must ever directly contain an ambient named b" is satisfied by P if it has a type $\langle X \mid G \rangle$ such that G does not contain a sequence $X_1 \xrightarrow{\mathsf{amb}\,a} X_2 \xrightarrow{\mathsf{amb}\,b} X_3$.

PROPOSITION 18 *Every term P has a type (although the type may contain* $\bullet$ *and thus not prove that the term "cannot go wrong").*

Our notion of types is very expressive — it *allows* a very fine-grained approximation to important questions. However, it is not known whether principal types always exist; we have neither proved nor disproved this. Thus, we now define a syntactically restricted type system for which we *do* prove that principal types exist.

4.3 Modest and discrete types; existence of principal types

DEFINITION 19 *Define the relation* $=_{(\leq)}$ *on prefix types as the least equivalence relation that contains* $\leq$.

DEFINITION 20 *Define the* ***stratification*** *function* $\mathbf{S}$ *by*

$$\mathbf{S}((\vec{a})) = \mathbf{S}(\langle\vec{\mu}\rangle) = 3 \qquad \mathbf{S}(\mathsf{amb}\,a) = 2 \qquad \mathbf{S}(C) = 1 \text{ when } C \neq \mathsf{amb}\,a$$

DEFINITION 21 *The shape graph G is* ***modest*** *iff for each* π, *one of the following conditions hold:*

1. ***Finite depth.*** *There is a number* n_π *such that whenever G contains a chain* $X_0 \xrightarrow{\pi_1} X_1 \xrightarrow{\pi_2} \cdots \xrightarrow{\pi_k} X_k$ *with every* $\mathbf{S}(\pi_i) \leq \mathbf{S}(\pi)$, *there are at most* n_π *different i's such that* $\pi_i =_{(\leq)} \pi$.
2. ***Monomorphic recursion.*** *Whenever G contains a chain* $X_0 \xrightarrow{\pi_1} X_1 \xrightarrow{\pi_2} \cdots \xrightarrow{\pi_k} X_k$ *with every* $\mathbf{S}(\pi_i) \leq \mathbf{S}(\pi)$ *and* $\pi_1 =_{(\leq)} \pi =_{(\leq)} \pi_k$, *then* $X_1 = X_k$.

DEFINITION 22 *The shape graph G is* ***discrete*** *iff both of these hold:*

1. *For each capability C that is not* amb *a for some a, whenever G contains a chain* $X_0 \xrightarrow{C} X_1 \xrightarrow{C} \cdots \xrightarrow{C} X_k$ *of edges all decorated with C and any two of the* X_i*'s are identical, then* $X_0 = X_1 = \cdots = X_k$.
2. *G does not contain any message type of the shape* $\{C_1, \ldots, C_k\}*$ *such that one of the* C_i*'s is* amb *a.*

Allowing only modest *and* discrete types yields **principal typings** (defined in [21]):

THEOREM 23 *For every term P which has at least one modest discrete type, there is a modest discrete type* τ *that is minimal among P's modest discrete types.*

The restriction to modest discrete type may feel somewhat artificial; indeed these properties have been designed specifically to allow the theorem to hold. While it is easy to construct terms where non-modest types allow a more precise analysis, they do not seem to correspond to natural programming styles. We conjecture that the restriction of expressive power entailed by requiring modesty and discreteness does not seriously impede PolyA's ability to analyse real-world software designs.

The proof of Theorem 23 is non-constructive and does not point to an effective procedure for *finding* a principal type. In [14] we have defined (and implemented) a practical type inference algorithm for a yet more restricted version of PolyA, but its principality properties are not yet well understood.

Requiring discreteness of types loses Prop. 18: There exist terms having no discrete type. However, all (ν-free) terms of the original ambient calculus have types:

PROPOSITION 24 *Any term P that does not contain* amb *a inside* $\langle\vec{M}\rangle$ *has a modest discrete type, and so also a principal such.*

5 Extended and modified ambient calculi

Our framework is strong enough to handle many ambient calculus variants with different reduction rules. In most cases, PolyA can be extended to deal with such variation simply by adjusting Defn. 14 with conditions systematically derived from the changed or new reduction rules. If this is done correctly and the new or changed rules are straightforward rewriting steps, then it is simple to extend the proof of Thm. 16. The rest of our theory will then carry through unchanged, including the existence of principal types. We illustrate this principle with examples of such extensions.

Boxed Ambients [4] removes the open capability; instead processes can communicate across ambient boundaries with directional communication actions:

$$\textbf{Prefixes: } p ::= M \quad \langle\vec{M}\rangle^{\uparrow} \quad \langle\vec{M}\rangle^{\star} \quad \langle\vec{M}\rangle^{\downarrow a} \quad (\vec{a})^{\uparrow} \quad (\vec{a})^{\star} \quad (\vec{a})^{\downarrow a}$$

There are corresponding reduction rules such as:

$$\overline{\langle\vec{M}\rangle^{\downarrow b}.P \mid b[Q \mid (\vec{a})^{\star}.R] \hookrightarrow P \mid b[Q \mid [a_i \mapsto M_i]_i R]}$$

Our prefix type syntax is easily extended to include the new actions. The new reduction rules can be used to derive local closure conditions such as:

$$\{(X_0 \xrightarrow{\langle\mu_1,\ldots,\mu_k\rangle^{\downarrow b}} X), (X_0 \xrightarrow{\text{amb } b} Y), (Y \xrightarrow{(a_1,\ldots,a_k)^{\star}} Z)\} \subseteq G$$
$$\Rightarrow \langle X \mid G\rangle \leq \langle X_0 \mid G\rangle \wedge [a_l \mapsto \mu_l]_{1\leq l\leq k}\langle Z \mid G\rangle \leq \langle Y \mid G\rangle$$

Safe Ambients [12] introduces *co-capabilities* where both interaction parties must present a capability. The reduction rules are amended to require this, e.g.:

$$\overline{a[\overline{\text{open}}\ a.P \mid Q] \mid \text{open } a.R \hookrightarrow P \mid Q \mid R}$$

It is straightforward to extend PolyA to systems with co-capabilities. For example, condition 3 of Defn. 14 would be replaced by:

$$\{(X_0 \xrightarrow{\text{amb } a} X), (X_0 \xrightarrow{\text{open } a} Y), (X \xrightarrow{\overline{\text{open}}\ a} Z)\} \subseteq G$$
$$\Rightarrow \langle X \mid G\rangle \leq \langle X_0 \mid G\rangle \wedge \langle Y \mid G\rangle \leq \langle X_0 \mid G\rangle \wedge \langle Z \mid G\rangle \leq \langle X_0 \mid G\rangle.$$

The **M**^{3} calculus [10] introduces a new method of inter-ambient communication; a new capability to can move a process into a neighbour ambient:

$$\overline{a[P \mid \text{to } b.Q] \mid b[R] \hookrightarrow a[P] \mid b[Q \mid R]}$$

This, too, is easily expressed as a closure condition:

$$\{(X_0 \xrightarrow{\text{amb } b} X), (X_0 \xrightarrow{\text{amb } a} Y), (Y \xrightarrow{\text{to } b} Z)\} \subseteq G$$
$$\Rightarrow \langle Z \mid G\rangle \leq \langle X \mid G\rangle$$

References

[1] 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 Technical Report BUCS-TR-2000-021, Comp.Sci. Department, Boston University, 2000.

[2] T. Amtoft, A. J. Kfoury, S. M. Pericas-Geertsen. Orderly communication in the ambient calculus. *Computer Languages,* 28, 2002.

[3] T. Amtoft, H. Makholm, J. B. Wells. PolyA: True type polymorphism for Mobile Ambients. Technical Report HW-MACS-TR-0015, Heriot-Watt Univ., School of Math. & Comput. Sci., 2004.

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

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

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

[8] L. Cardelli, P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys,* 17(4), 1985.

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

[10] M. Coppo, M. Dezani-Ciancaglini, E. Giovannetti, I. Salvo. M3: Mobility types for mobile processes in mobile ambients. In *CATS 2003,* vol. 78 of *ENTCS,* 2003.

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

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

[13] C. Lhoussaine, V. Sassone. A dependently typed ambient calculus. In *Programming Languages & Systems, 13th European Symp. Programming,* vol. 2986 of *LNCS.* Springer-Verlag, 2004.

[14] H. Makholm, J. B. Wells. Type inference for PolyA. Technical Report HW-MACS-TR-0013, Heriot-Watt Univ., School of Math. & Comput. Sci., 2004.

[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. *Nordic Journal of Computing,* 8, 2001. A preliminary version appeared at POPL'00.

[18] B. C. Pierce, D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM,* 47(3), 2000.

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

[20] D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation.* PhD thesis, University of Edinburgh, 1995. Report no ECS-LFCS-96-345.

[21] J. B. Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming,* vol. 2380 of *LNCS.* Springer-Verlag, 2002.

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