Instant Polymorphic Type Systems for Mobile Process Calculi: Just Add Reduction Rules and Close*

Henning Makholm and J. B. Wells

Heriot-Watt University

Abstract. Many different *mobile process calculi* have been invented, and for each some number of type systems has been developed. Soundness and other properties must be proved separately for each calculus and type system. We present the *generic* polymorphic type system Poly✶ which works for a wide range of mobile process calculi, including the π-calculus and Mobile Ambients. For any calculus satisfying some general syntactic conditions, well-formedness rules for types are derived automatically from the reduction rules and Poly✶ works otherwise unchanged. The derived type system is automatically sound (i.e., has subject reduction) and often more precise than previous type systems for the calculus, due to Poly✶'s *spatial polymorphism*. We present an implemented type inference algorithm for Poly✶ which automatically constructs a typing given a set of reduction rules and a term to be typed. The generated typings are *principal* with respect to certain natural type shape constraints.

1 Introduction

Many calculi that intend to capture the essence of *mobile* and *distributed* computing have been invented. The most well-known of these are probably the π-calculus [18] and the *Mobile Ambients* calculus (MA) by Cardelli and Gordon [8], but they haveinspired the subsequent development of a wide variety of variants and alternatives, which are variously argued to be easier to program in or reason about, and/or closer to some operational intuition about how programs in a mobile, distributed setting can be implemented. The field stays productive; new calculi are still being proposed and there is not a clear consensus about what should be considered *the* fundamental mobility calculus.

The majority of these calculi share the basic architecture of MA: They borrow from the π-calculus the syntactic machinery for talking about sets of parallel, communicating processes, plus its primitive operator ν for generating unique names. To this they add some kind of *spatial structure*, usually in the form of a tree of locations where processes can reside. The tree can generally evolve under program control as the processes in it execute; the different calculi provide quite different primitives for mutating it. Mobility calculi also provide for *communication* between processes that are near each other, usually modelled on the communication primitive of the π-calculus, but again with variations and often extended with the possibility to communicate "capabilities", "paths", or other restricted pieces of process syntax, rather than just names.

Most process calculi have an associated *type system*, either one that was designed

* Supported by EC FP5/IST/FET grant IST-2001-33477 "DART", and Sun Microsystems equipment grant EDUD-7826-990410-US.

with the calculus from the beginning, or one that was retrofitted later. These type systems are closely tied to a specific calculus and its particular primitives. Once a type system has been designed and its properties (such as soundness or the applicability of a particular type inference algorithm) have been proved, it is in general not trivial to see whether these properties will survive changes to the calculus.

1.1 A generic type system. In contrast, this paper presents the *generic* type system Poly✶ which works for a wide range of mobile process calculi. To use Poly✶, one simply instantiates it with the reduction rules that specify the semantics of the target calculus's primitives. From this, a set of provably sound well-formedness rules for types can be *mechanically* produced, guaranteeing that types that satisfy the rules are sound with respect to the reduction rules, i.e., subject reduction holds. The reduction rules can also be used to guide an automatic *type inference* algorithm for the instantiated type system. The inference algorithm produces a type which is *principal* with respect to certain natural constraints on the shape of types. Our implementation offers several possibilities for tuning the *precision* of the type system it implements, but the use of these is optional — it will always produce a typing even when given only the raw reduction rules of the target calculus.

For this to work, the target calculus must make one small concession to Poly✶, namely that its *syntax* is sufficiently regular that the implementation can make sense of its terms and reduction rules. We define a **metacalculus** Meta✶ which gives a syntax that is easy to parse and manipulate, while flexible enough that many calculi can be viewed as instances of it without deviating much from their native notations. Meta✶ does not include any fixed *semantics* except for the usual semantics of parallelism and name restriction, but instead provides a common notion of substitution and a notation for rewriting rules that fits how semantics for process calculi are usually defined.

1.2 Poly✶'s relation to other reasoning principles. A long-term goal of Poly✶ is to make it possible to view many previously existing mobility calculi type systems as instances of Poly✶, at least with regards to using the type system to statically verify that certain bad behaviours do not occur. The design we present here does not quite reach that point; there are features of existing type systems that we have not yet incorporated in Poly✶. We believe it will be particularly important to express some form of the *single-threaded* locations introduced by the original type system for Safe Ambients [16].

We do not expect actual programming environments based on mobility calculi to use the fully general Poly✶ formalism as their type discipline. Considerations of performance and integration will generally dictate that production environments instead use hand-crafted specialised type systems for the language they support, though *ideas* from Poly✶ may well be employed.

A generic implementation of Poly✶, such as the one we present here, should be a valuable tool for *exploring the design space* for mobility calculi in general. It will make it easy to change some aspect of one's rewriting rules, try to analyse some terms, and see which effect the new rules have on, for example, the interference-control properties of one's calculus. At the same time, our Poly✶ implementation makes it easy to experiment with exactly how strong a type system one wants to use in practice, because our

implementation supports tuning the precision of types in very small steps.[1]

Like every nontrivial type system with an inference algorithm, Poly✶ can be used as a *control/data flow analysis* to provide the substratum for more specialised automatic *program analyses*.[2] (Readers who are uncomfortable about applying the term "type system" to Poly✶ are invited to think "versatile program analysis framework" each time we write "type system".) However, we have no pretension of subsuming all other analysis techniques for mobility or process calculi in general. Process calculi have provided the setting for many advanced techniques for reasoning about, for example, behavioural equivalence of processes. Poly✶ does not claim to compete with these.

1.3 Spatial polymorphism. The Poly✶ type system descends from (but significantly generalises and enhances) our earlier work [2] on PolyA, a polymorphic type system specific to Mobile Ambients. It inherits from PolyA the difference from most other type systems for mobility calculi that the emphasis is on types for *processes* rather than types for (ambient or channel) *names*.[3] In fact, types for names have completely vanished: A name has no intrinsic type of its own, but is distinguished solely by the way it can be used to form processes.

Poly✶ works by approximating the set of terms a given term can possibly evolve to using the given reduction rules. Its central concept is that of a **shape predicate** which is an automaton that describes a set of process terms. Shape predicates that satisfy certain well-formedness rules are **types**. These rules are derived from the reduction rules of the target calculus and guarantee that the set of terms denoted by a type is closed under the reduction relation, i.e., *subject reduction* holds.

This design gives rise to a new (with PolyA) form of polymorphism that we call **spatial polymorphism**. The type of a process may depend on where in the spatial structure it is found. When the process moves, it may come under influence of another part of the type which allows more reductions. For example, consider a calculus which has the single reduction rule $\mathtt{a}[\mathsf{eat}\ \mathtt{b} \mid \mathtt{P}] \mid \mathtt{b}[\mathtt{Q}] \hookrightarrow \mathtt{a}[\mathtt{P} \mid \mathtt{b}[\mathtt{Q}]]$. In this calculus, the term $\mathtt{x}[\mathsf{eat}\ \mathtt{z1} \mid \mathsf{eat}\ \mathtt{z2}] \mid \mathtt{y1}[\mathsf{eat}\ \mathtt{x} \mid \mathtt{z1}[0]] \mid \mathtt{y2}[\mathsf{eat}\ \mathtt{x} \mid \mathtt{z2}[0]]$ has a Poly✶ type, shown in Figure 1, that says that $\mathtt{x}[]$ may contain $\mathtt{z1}[]$ when it it inside $\mathtt{y1}[]$, or $\mathtt{z2}[]$ when it it inside $\mathtt{y2}[]$, but can contain neither when it is found at the top level of the term. Thus Poly✶ can prove that the term satisfies the safety policy that z1 and z2 may never be found side by side. To our knowledge, type systems based on earlier paradigms cannot do this.

With spatial polymorphism, *movement* is what triggers the generation of a polymorphic variant of the original analysis for a piece of code. This is different from, and orthogonal to, the more conventional form of name-parametric polymorphism in the polymorphic π-calculus [24], where it is *communication* that causes polymorphic variants to be created. Poly✶ does not support the latter form of polymorphism (and neither

[1] These fine tuning options are omitted from this paper due to lack of space, but they are described in detail in the implementation's documentation.

[2] Indeed it is well known [20, 3] that the difference between an advanced flow analysis and an advanced type system is often just a question of different perspectives on the same underlying machinery. The presentation of Poly✶ is closer to the data-flow viewpoint than is common for type systems, though this of course does not make Poly✶ any less a type system.

[3] There are a number of type systems for process calculi *without* an explicit notion of locations which assign types to processes rather than names, for example [4, 14, 27, 11].

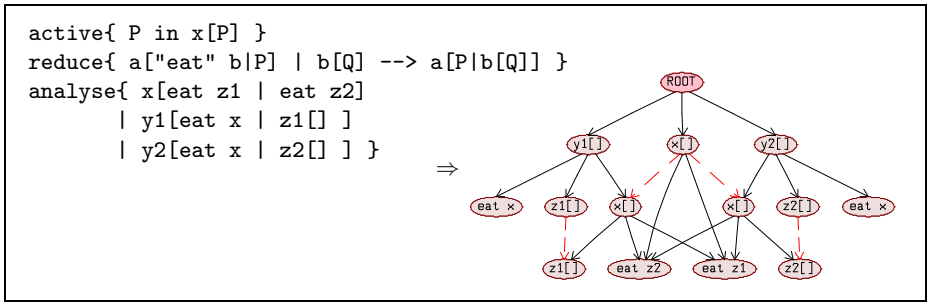

Fig. 1. *Input to our type inference implementation for analysing a term in the fictional "eat calculus", and the inferred type graph as rendered by the VCG graph layout tool [23] (the dashed lines represent subtyping edges).*

does any type system for a mobility calculus with explicit locations that we are aware of); we leave it to future work to try to combine the strengths of these two principles.

1.4 Notation and preliminaries. $\boxed{X}$, where X is any metavariable symbol, stands for the set that X ranges over. $\mathcal{P}_{\text{fin}}(A)$ is the set of finite subsets of the set A. $A \xrightarrow{\text{fin}} B$ is the set of finite partial maps from A to B. Dom f is the set of x's such that $f(x)$ is defined. In contexts where a sequence of similar objects are indexed with indexes up to k, it is to be understood that k can be any integer ≥ 0. Thus, if the first index is 0, the sequence must have at least one element; sequences indexed from 1 to k may be empty.

2 Meta✶: A metacalculus of concurrent processes

The metacalculus Meta✶ defined in this section is the *syntactic* setting for Poly✶. Its role is to let us present the generic properties of Poly✶ without resorting to handwaving. Though we define a reduction relation and some other formal properties for Meta✶, these exist solely as support for making formal statements about Poly✶. We do not intend Meta✶ to take the place of any existing calculi or frameworks.

As a first approximation, Meta✶ is a "syntax without a semantics" except that it does give semantics to a few basic constructs, e.g., process replication and substitution.

2.1 Terms. Figure 2 shows the syntax of process terms in Meta✶. The trivial process 0, parallel composition of processes $P \mid Q$, process replication $!P$, and name restriction $\nu(x).P$ are all well-known from most process calculi, including π-calculus and MA. They are given their usual behaviour by the structural congruence relation $\equiv$.

Meta✶ amalgamates all other process constructors into the general concept of a **form**. Forms have no intrinsic meaning until a set of reduction rules give them one. Examples of forms include the communication actions "$x\langle y\rangle$" and "$x(y)$" from the π-calculus, the movement capabilities "in x", "out x", and "open x" from Mobile Ambients, and even ambient boundaries themselves, which we write as "x []". We support the traditional syntax "$x[P]$" for ambients by interpreting "$E_1 \ldots E_k[P]E'_1 \ldots E'_{k'}$" as syntactic sugar for "$E_1 \ldots E_k$ [] $E'_1 \ldots E'_k.P$". Except for this syntactic convention, the symbol [] has no special interpretation in Meta✶ and it is a (single) name just like in and

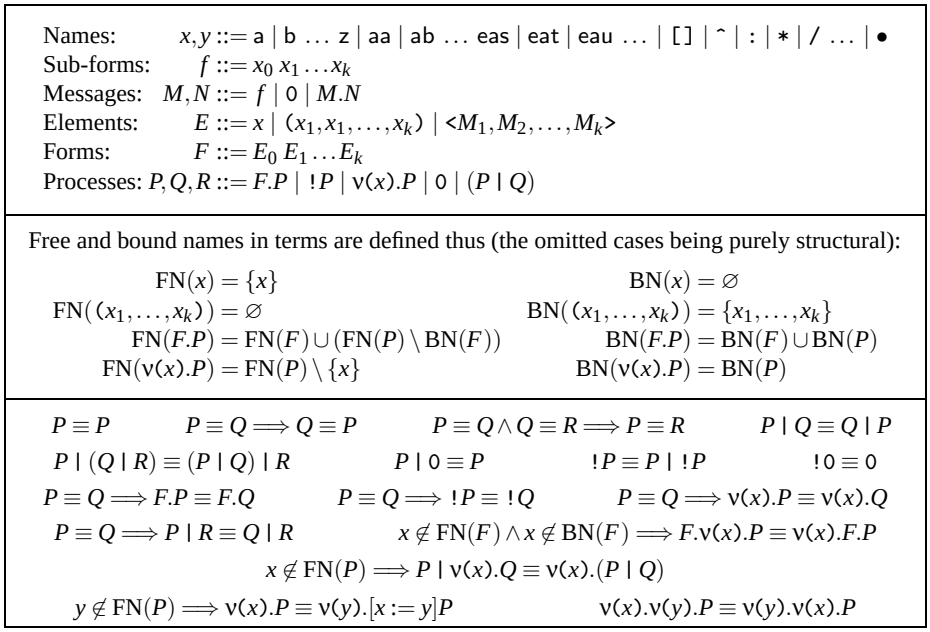

Fig. 2. *Syntax of Meta✶ plus its structural congruence relation*

out. The process $F.\mathsf{0}$ can be abbreviated as F.

A form consists of a nonempty sequence of **elements**, each of which is either a **name**, a **binding** element, or a **message** element. Names are used to name channels, ambients, and so on, but also work as keywords that distinguish forms with different roles in the calculus. A keyword is simply a free name that is matched explicitly by some reduction rule. Most non-alphanumeric ASCII characters that do not have any special meaning (`^`, `:`, `*`, `/`, etc.) are also names and so can be used as keywords. With these we can encode, e.g., annotated communication actions like "$\langle M \rangle^*$" or "$(x)^y$" from Boxed Ambients [5] using pseudo-TEX notation as the forms "$\mathsf{<}M\mathsf{>\,\hat{}\,*}$" and "$(x)\,\mathsf{\hat{}}\,y$".

Binding elements $(x_1,\ldots,x_k)$ are used to create forms that bind names in the process they are applied to. The canonical use of this is for constructing receive actions, but again the meaning of the form is specified only by the reduction rules. Message elements $\mathsf{<}\cdots\mathsf{>}$ allow a form to contain other forms, which — given appropriate reduction rules for communication — can later be substituted into processes. For technical reasons we have to restrict the forms contained in message elements in that they cannot contain message or binding elements themselves. We refer to such restricted forms and their elements as **sub-forms** and **sub-elements**. In future work we hope to be able to handle calculi such as the spi-calculus [1] which communicate structured messages.

It is not uncommon for calculi to prefer using an explicit recursion construction "$P ::= \mathbf{rec}\, X.P$" to express infinite behaviour rather than the process replication operator "`!`". There are certain technical problems with supporting this directly in Meta✶ (which may however be approachable by novel techniques involving regular grammars developed by Nielson et al. [19]). In the common case where the target calculus does not allow location boundaries to come between the $\mathbf{rec}\, X$ binder and the bound X, it can

$$\mathcal{S}^{\mathbf{E}}x = \begin{cases} x & \text{when } x \notin \mathrm{Dom}\,\mathcal{S} \\ y & \text{when } \mathcal{S}(x) = y \text{ for some } y \\ \bullet & \text{otherwise} \end{cases} \qquad \mathcal{S}^{\mathbf{M}}x = \begin{cases} \mathcal{S}(x) & \text{when } x \in \mathrm{Dom}\,\mathcal{S} \\ x & \text{otherwise} \end{cases}$$

$$\mathcal{S}^{\mathbf{P}}(\nu(x).P) = \nu(x).\mathcal{S}^{\mathbf{P}}P \qquad \mathcal{S}^{\mathbf{P}}(x.P) = \begin{cases} \mathcal{S}(x)_*(\mathcal{S}^{\mathbf{P}}P) & \text{when } x \in \mathrm{Dom}\,\mathcal{S} \\ x.(\mathcal{S}^{\mathbf{P}}P) & \text{otherwise} \end{cases}$$

$$(M.N)_*P = M_*(N_*P) \qquad \mathsf{0}_*P = P \qquad f_*P = f.P$$

Fig. 3. *The actions of term substitution. $\mathcal{S}^{\mathbf{M}}$ is the action on messages, $\mathcal{S}^{\mathbf{E}}$ the action on elements, $\mathcal{S}^{\mathbf{F}}$ on (sub)forms, and $\mathcal{S}^{\mathbf{P}}$ on processes. The omitted cases (including the one for $\mathcal{S}^{\mathbf{F}}$) simply substitute componentwise into the syntactic element in question. The M_*P helper operator serves to linearise messages once we do not need to keep track of whether they are composite or not. (In other systems, this is often done by the structural congruence relation instead.)*

easily be simulated in Meta✶ by adding the reduction rule $\mathsf{spawn\ a} \mid \mathsf{rec\ a.P} \hookrightarrow \mathsf{P}$ and then representing **rec** $X.\cdots.X$ as $\nu(\mathsf{x}).(\mathsf{spawn\ x} \mid \mathsf{!\,rec\ x.}\cdots.\mathsf{spawn\ x})$.

2.2 Well-scoped terms. The process term P is **well scoped** iff it contains no nested binding of the same name and none of its free names also appear bound in the term. Formally, it is required that (1) $\mathrm{BN}(P)$ and $\mathrm{FN}(P)$ are disjoint, (2) whenever P contains $F.Q$, $\mathrm{BN}(F)$ and $\mathrm{BN}(Q)$ are disjoint, and (3) whenever P contains $\nu(x).Q$, $x \notin \mathrm{BN}(Q)$.

We generally require that terms are always well scoped. The reduction rules in an instantiation of Meta✶ must preserve well-scopedness. This simplifies the type analysis because then we do not have to support α-conversion of ordinary binding elements.

We must still handle α-conversion of private names, which is built into the $\equiv$ relation, but we will assume that it is not used to create terms that are not well scoped.

2.3 Substitutions. Substitutions in Meta✶ substitute *messages* for *names*. The fact that entire *processes* cannot be substituted is an important technical premise of Poly✶; it means that substitution can preserve well-scopedness. It is remarkable that mobility calculi in general refrain from substituting processes; calculi such as Seal [25] and $\mathbf{M^3}$ [12] which allow exchange of entire processes do it by local *movement* rather than *substitution*. This probably reflects the intuition that a running process is harder to distribute across a recipient process than a mere name or code sequence.

A (term) **substitution** $\mathcal{S}$ is a finite map from names to messages. Figure 3 defines the action of $\mathcal{S}$ on the various syntactic classes of Meta✶. In Mobile Ambients and its descendant calculi, the value exchanged in a communication operation can be either a name or a (sequence of) capabilities. The former is the case in reduction $\langle \mathsf{b} \rangle \mid (a).\mathsf{out}\,a.\mathsf{0} \hookrightarrow \mathsf{out\,b.0}$ and the latter in $\langle \mathsf{in\,b} \rangle \mid (a).\mathsf{x}[a.\mathsf{in\,c.0}] \hookrightarrow \mathsf{x}[\mathsf{in\,b.in\,c.0}]$. To support this, Fig. 3 contains special cases for the syntactic cases $M ::= F$ and $P ::= F.P$ when the form F is a lone name. In that case the substitution for the name is inserted directly into the message (or process structure).

In cases like $\{\mathsf{a} \mapsto \mathsf{b}\}^{\mathbf{P}}\mathsf{x}[\mathsf{out\ a.0}]$ where the substituted name occurs properly inside a form, the substitution is carried out componentwise for each form element, and the name is replaced in the rule for $\mathcal{S}^{\mathbf{E}}x$. In this context the replacement must be a name

too. This will be false if the term tries to reduce as $\langle \mathsf{in}\,\mathtt{b}\rangle \mid (a).\mathsf{out}\,a.0 \hookrightarrow \mathsf{out}\,(\mathsf{in}\,\mathtt{b}).0$. The published formalisms of most ambient-inspired calculi usually regard "$\mathsf{out}\,(\mathsf{in}\,\mathtt{b})$" as *syntactically* possible but *semantically* meaningless. That this configuration cannot occur is often the most basic soundness property of type systems for such calculi.

In Meta✶ such a semantic error becomes a syntactic one: It is simply not possible to use an entire form as an element (except indirectly through a message element). If, at runtime, a substitution nevertheless tries to do so, we substitute the special name "•", which is to be interpreted as, "an erroneous substitution happened here". Thus, with the MA communication rule, Meta✶ reduces <in b>.0 | (a).out a.0 ↪ out•.0. This convention is technically convenient because it allows us to bound the nesting depth of forms (using the sub-form restriction). Because most published calculi attach no semantics to forms like "$\mathsf{out}\,(\mathsf{in}\,\mathtt{b})$", we do not lose any real expressiveness.

Forms that contain • are inert in Meta✶ unless there are reduction rules that explicitly match on •. The calculus designer can also define reduction rules that create •'s in other situations to mark reduction results as "erroneous". For example, in the polyadic π-calculus, it is usually considered a run-time error if someone tries to send an m-tuple on a channel where another process is listening for an n-tuple, with $n \neq m$. By writing explicit rules[4] for such situations, they can be handled in parallel with malformed substitutions. (One cannot straightforwardly write patterns to test for malformed substitutions, which is one reason for building the generation of • into Meta✶).

In either case, the Poly✶ type system will conservatively estimate *whether* (and where) a • can occur. Which conclusions to draw from this (e.g., rejecting the input program due to "type error") is up to the designer of the calculus.

The definitions in Figure 3 do not worry about name capture. In general, therefore, $\mathcal{S}^{\mathbf{X}}X$ is only intuitively correct if $\mathrm{BN}(X)$ is disjoint from the names mentioned in $\mathcal{S}$. In practise, this will always follow from the assumption that all terms are well scoped.

2.4 Reduction rules. Figure 4 defines most of the syntax and semantics of reduction rules for Meta✶. Our full theory (and implementation) allows a slightly more expressive template language to the right of the arrow in **reduce** rules, but the subset we present here is sufficient to express the calculi listed in Sect. 2.5.

As an example, with this syntax we can describe Mobile Ambients by the ruleset

$$\begin{aligned}\mathcal{R}_{\mathrm{MA}} = \{\ & \textbf{active}\{\mathtt{P}\ \textbf{in}\ \mathtt{a}[\mathtt{P}]\},\\ & \textbf{reduce}\{\mathtt{a}[\mathsf{in}\,\mathtt{b}.\mathtt{P} \mid \mathtt{Q}] \mid \mathtt{b}[\mathtt{S}] \hookrightarrow \mathtt{b}[\mathtt{a}[\mathtt{P} \mid \mathtt{Q}] \mid \mathtt{S}]\},\\ & \textbf{reduce}\{\mathtt{a}[\mathtt{b}[\mathsf{out}\,\mathtt{a}.P \mid \mathtt{Q}] \mid \mathtt{S}] \hookrightarrow \mathtt{a}[\mathtt{S}] \mid \mathtt{b}[\mathtt{P} \mid \mathtt{Q}]\},\\ & \textbf{reduce}\{\mathsf{open}\,\mathtt{a}.\mathtt{P} \mid \mathtt{a}[\mathtt{R}] \hookrightarrow \mathtt{P} \mid \mathtt{R}\},\\ & \textbf{reduce}\{\texttt{<M>}.\mathtt{P} \mid \texttt{(a)}.\mathtt{Q} \hookrightarrow \mathtt{P} \mid \{\mathtt{a} := \mathtt{M}\}\mathtt{Q}\}\ \}\end{aligned}$$

These five rules are all that is necessary to instantiate Meta✶ to be Mobile Ambients.[5] The four **reduce** rules directly correspond to the reduction axioms of the target calculus. The rule **active**$\{\mathtt{P}\ \textbf{in}\ \mathtt{a}[\mathtt{P}]\}$ is the Meta✶ notation for the "evaluation context" rule

[4] E.g., **reduce**$\{\texttt{<M1,M2>}.\mathtt{P} \mid \texttt{(x1,x2,x3)}.\mathtt{Q} \hookrightarrow \bullet.0\}$ for $(m,n) = (2,3)$. Our implementation provides an extension for writing a single rule that catches all pairs (m,n) at once.

[5] The rules are not sufficient to get communication reduction with arbitrary arity. Our implementation provides a syntax for defining arbitrary-arity communication rules, but for reasons of space and clarity we omit it in our formal development.

$$
\begin{array}{lll}
\text{Name variables:} & \mathring{x} ::= \mathtt{a} \mid \mathtt{b} \mid \mathtt{c} \mid \cdots & \\
\text{Message variables:} & \mathring{m} ::= \mathtt{M} \mid \mathtt{N} \mid \cdots & \\
\text{Process variables:} & \mathring{p} ::= \mathtt{P} \mid \mathtt{Q} \mid \mathtt{R} \mid \cdots & \\
\text{Substitutes:} & \underline{s} ::= \mathring{x} \mid \mathring{m} & \\
\text{Element templates:} & \underline{E} ::= \mathring{x} \mid x \mid \mathtt{(}\mathring{x}_1,\ldots,\mathring{x}_k\mathtt{)} \mid \mathtt{<}\mathring{m}_1,\ldots,\mathring{m}_k\mathtt{>} & \\
\text{Forms templates:} & \underline{F} ::= \underline{E}_0\,\underline{E}_1\ldots\underline{E}_k & \\
\text{Process templates:} & \underline{P} ::= \mathring{p} \mid \underline{F}.\underline{P} \mid \mathtt{0} \mid (\underline{P}_1 \mid \underline{P}_2) & \\
 & \quad \mid \{\mathring{x}_0 := \underline{s}_0,\ldots,\mathring{x}_k := \underline{s}_k\}\ \mathring{p} & (\mathsf{R}) \\
\text{Rules:} & \mathcal{R}^1 ::= \mathbf{reduce}\{\underline{P}_1 \hookrightarrow \underline{P}_2\} \mid \mathbf{active}\{\mathring{p}\ \mathbf{in}\ \underline{P}\} & \\
\text{Rulesets:} & \mathcal{R} \in \mathcal{P}_{\text{fin}}(\boxed{\mathcal{R}^1}) &
\end{array}
$$

The syntactic choice marked (R) is allowed only in a **reduce** rule to the *right* of the arrow.

Let an **term instantiation** $\mathcal{V}$ map $\boxed{\mathring{x}}$ to $\boxed{x} \setminus \{\bullet\}$, $\boxed{\mathring{m}}$ to $\boxed{M}$, and $\boxed{\mathring{p}}$ to $\boxed{P}$. It applies to templates strictly componentwise, except for the case that fills in and applies a substitution:

$$\mathcal{V}^{\mathbf{P}}(\{\ldots,\mathring{x}_i := \underline{s}_i,\ldots\}\ \mathring{p}) = \{\ldots,\mathcal{V}(\mathring{x}_i) \mapsto \mathcal{V}(\underline{s}_i),\ldots\}^{\mathbf{P}}(\mathcal{V}(\mathring{p}))$$

As a special exception, $\mathcal{V}^{\mathbf{P}}\underline{P}$ is considered *undefined* if $\mathcal{V}(\mathring{x}_1) = \mathcal{V}(\mathring{x}_2)$ for $\mathring{x}_1 \neq \mathring{x}_2$ such that $\mathring{x}_1$ occurs in $\underline{P}$ below a form template containing a binding element $\mathtt{(}\ldots,\mathring{x}_2,\ldots\mathtt{)}$.
For example, $\{\mathtt{a} \mapsto \mathtt{x}, \mathtt{b} \mapsto \mathtt{x}, \mathtt{c} \mapsto \mathtt{x}\}$ cannot be applied to `(a).c.0 | (b).c.0`, which would otherwise capture names and produce `(x).x.0 | (x).x.0`.

$$\frac{\mathbf{reduce}\{\underline{P}_1 \hookrightarrow \underline{P}_2\} \in \mathcal{R}}{\mathcal{R} \vdash \mathcal{V}^{\mathbf{P}}\underline{P}_1 \hookrightarrow \mathcal{V}^{\mathbf{P}}\underline{P}_2} \qquad \frac{\mathcal{R} \vdash P \hookrightarrow Q}{\mathcal{R} \vdash \nu(x).P \hookrightarrow \nu(x).Q}$$

$$\frac{\mathbf{active}\{\mathring{p}\ \mathbf{in}\ \underline{P}\} \in \mathcal{R} \qquad \mathcal{R} \vdash P \hookrightarrow Q}{\mathcal{R} \vdash (\mathcal{V}[\mathring{p} \mapsto P])^{\mathbf{P}}\underline{P} \hookrightarrow (\mathcal{V}[\mathring{p} \mapsto Q])^{\mathbf{P}}\underline{P}} \qquad \frac{\mathcal{R} \vdash P \hookrightarrow Q}{\mathcal{R} \vdash P \mid R \hookrightarrow Q \mid R} \qquad \frac{P \equiv Q \qquad \mathcal{R} \vdash Q \hookrightarrow R}{\mathcal{R} \vdash P \hookrightarrow R}$$

Fig. 4. *Syntax and semantics of reduction rules.*

$P \hookrightarrow P' \Longrightarrow a[P] \hookrightarrow a[P']$. This is, in fact, the only concrete **active** rule that we have so far needed for encoding existing calculi. We might just have hard-coded something like this rule into Meta✶, but we find it cleaner not to have any built-in distinction between "action" forms and "process container" forms in the theory.

The lower half of Figure 4 defines how to derive a reduction relation between process terms from a ruleset. For example, let $\mathcal{R}_{\text{eat}}$ be the ruleset for the fictional calculus from Fig. 1: $\mathcal{R}_{\text{eat}} = \{\ \mathbf{active}\{\mathtt{P}\ \mathbf{in}\ \mathtt{a}[\mathtt{P}]\}, \mathbf{reduce}\{\mathtt{a}[\mathtt{eat\,b} \mid \mathtt{P}] \mid \mathtt{b}[\mathtt{Q}] \hookrightarrow \mathtt{a}[\mathtt{P} \mid \mathtt{b}[\mathtt{Q}]]\}\ \}$. We can then instantiate the first inference rule in the bottom third of Fig. 4 to obtain

$$\mathcal{R}_{\text{eat}} \vdash \mathtt{y1}[\mathtt{eat\ x} \mid \mathtt{z1}[\mathtt{0}]] \mid \mathtt{x}[\mathtt{eat\ z1} \mid \mathtt{eat\ z2}] \hookrightarrow \mathtt{y1}[\mathtt{x}[\mathtt{eat\ z1} \mid \mathtt{eat\ z2}] \mid \mathtt{z1}[\mathtt{0}]]$$

by choosing $\mathcal{V}$ to be $\{\mathtt{a} \mapsto \mathtt{y1}, \mathtt{b} \mapsto \mathtt{x}, \mathtt{P} \mapsto \mathtt{z1}[\mathtt{0}], \mathtt{Q} \mapsto (\mathtt{eat\ z1} \mid \mathtt{eat\ z2})\}$.

A reduction rule must not allow a well-scoped term to reduce to a non-well-scoped one. In order to guarantee this, the process templates in them must satisfy some scoping restrictions that are not apparent from the syntax. The restrictions will be satisfied by most rules that are intuitively sensible; because a precise understanding of how the restrictions work is not important for a high-level understanding of Meta✶, we refer to this paper's long version [17] for a precise definition.

2.5 Example instantiations. We have checked (using machine-readable rulesets for our type inference implementation for Meta✶/Poly✶) that Meta✶ can handle π-calculus [18]; Mobile Ambients [8]; Safe Ambients [16] and various variants regarding where the $\overline{\mathsf{out}}$ capability must be found and which name co-capabilities must refer to (variants with anonymous co-capabilities also exist [15]); the Seal calculus [25] in the non-duplicating variation of [10]; Boxed Ambients [5], as well as its "asynchronous" and "Seal-like" variants (the latter being what later papers most often refer to as BA); Channelled Ambients [21]; NBA [6]; Push and Pull Ambient Calculus [22]; and $\mathbf{M^3}$ [12].

In many of these cases, Meta✶ supports the straightforward way to notate process terms as flat ASCII text, but in some cases the native punctuation of the target calculus must be changed superficially to conform to Meta✶ conventions about how a form looks. For example, the original send action $\bar{y}x$ from [18] is represented as "*y*<*x*>" (but "/*y x*" would also have worked), and "$\overline{\mathsf{enter}}(x,y)$" from [6] becomes "co-enter$(x)y$", because it binds *x* in its continuation but uses *y* to handshake with the entering ambient. The "$n[c_1,\ldots,c_k;P]$" construction in Channelled Ambients [21] can be represented as "$n[\mathsf{cs}.(c_1.0 \mid \cdots \mid c_k.0) \mid \mathsf{ps}.P]$". In our ruleset for Mobile Ambients with Objective Moves [7], the fact that reduction rules cannot inspect the structure of messages forces us to represent the original "*go M.m*[*P*]" as "go.*M*.*m*[*P*]".

3 Poly✶: Types for Meta✶

3.1 Shape predicates. As described in the introduction, *shape predicates* are the central concept in Poly✶. A shape predicate denotes a set of process terms; certain shape predicates that are provably closed under reduction are *types*. The full language of shape predicates is somewhat complex, so let us introduce it piecewise. The basic idea of shape predicates can be explained simply: *A shape predicate looks like a process term. It matches any process term that can arise by repeatedly duplicating and/or removing sub-terms of the shape predicate.* Here, "duplicating" and "removing" sub-terms means applying the rewriting rules $\pi \rightsquigarrow \pi \mid \pi$ and $\pi \rightsquigarrow 0$ to any syntactic subterm of the shape predicate, in addition to using the structural congruence relation for terms.

For example, a shape predicate written $\mathsf{a}[\mathsf{in}\,\mathsf{b} \mid \mathsf{in}\,\mathsf{c}] \mid \mathsf{c}[0]$ would match the terms $\mathsf{a}[\mathsf{in}\,\mathsf{b} \mid \mathsf{in}\,\mathsf{c}] \mid \mathsf{c}[0]$ (which is identical to the shape predicate) and $\mathsf{a}[\mathsf{in}\,\mathsf{b}] \mid \mathsf{a}[\mathsf{in}\,\mathsf{c}] \mid \mathsf{c}[0]$ (which arises by duplicating $\mathsf{a}[\cdots]$ and then removing one of the in subterms in each of the copies). But $\mathsf{a}[\mathsf{in}\,\mathsf{b}] \mid \mathsf{c}[\mathsf{a}[0]]$ does not match, because duplicating subterms cannot make a[] appear below a c[]. Neither is $\mathsf{in}\,\mathsf{b} \mid \mathsf{in}\,\mathsf{c} \mid \mathsf{c}[0]$ allowed — when removing the a[] form, the entire subterm below it must be removed.

The type in Fig. 1 can be written in term shape as $\mathsf{y1}\big[\mathsf{eat}\,\mathsf{x} \mid \mathsf{z1}[0] \mid \mathsf{x}[\mathsf{z1}[0] \mid \mathsf{eat}\,\mathsf{z1} \mid \mathsf{eat}\,\mathsf{z2}]\big] \mid \mathsf{x}[\mathsf{eat}\,\mathsf{z1} \mid \mathsf{eat}\,\mathsf{z2}] \mid \mathsf{y2}\big[\mathsf{x}\big[\mathsf{eat}\,\mathsf{z1} \mid \mathsf{eat}\,\mathsf{z2} \mid \mathsf{z2}[0]\big] \mid \mathsf{eat}\,\mathsf{x} \mid \mathsf{z2}[0]\big]$.

In practice shape predicates cannot be exactly term-shaped, but it pays to keep this naive idea in mind as an intuition about what shape predicates are. When we introduce complications in the rest of this subsection, they should all be understood as "whatever is necessary to make the naive idea work in practice".

Replication (!*P*) is ignored when matching shape predicates. This is sensible because !*P* behaves like an infinite number of *P*'s running in parallel, and any *finite* number of *P*'s in parallel match a shape predicate exactly if a single *P* does.

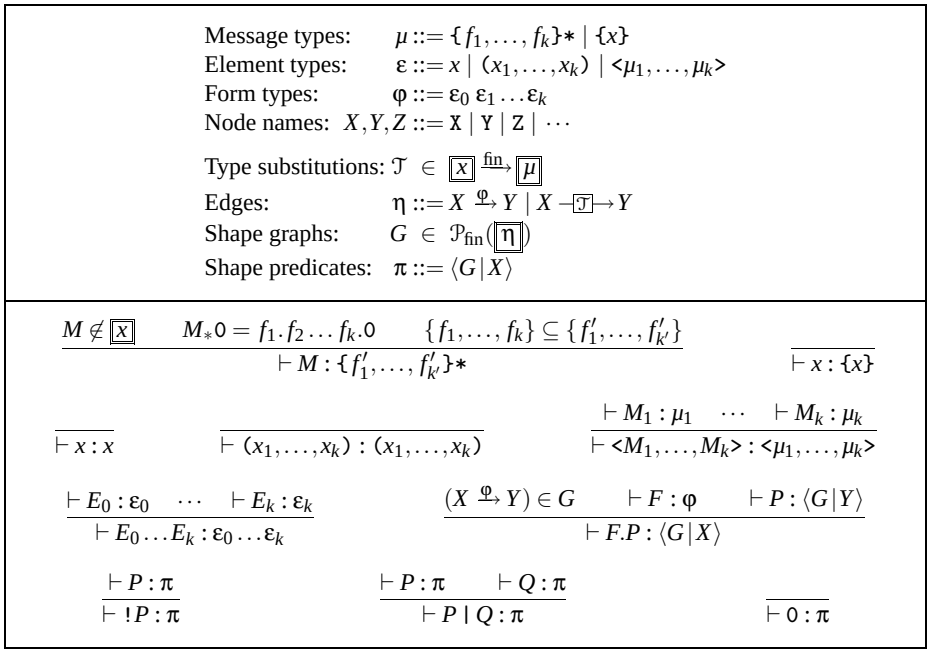

Fig. 5. *The syntax and semantics of shape predicates. Edges of the form $X \multimap\!\boxed{\mathcal{T}}\!\!\rightarrow Y$ do not influence the semantics of the shape predicate; Sect. 3.2 explains what they are for.*

We want to represent all possible computational future of each term smashed together in a single shape predicate. This creates problems for the naive idea, because terms such as `!x[eat x]` can evolve to arbitrary deep nestings of `x[···]`. Therefore we need shape predicates to be *infinitely* deep trees. We restrict ourselves to infinite shape predicates with finite *representations* — in other words, regular trees.

There are several known ways of representing regular trees as linear syntax, but we have found it easier to work directly with *graphs*. A shape predicate now has the form $\langle G \mid X\rangle$, where G is a directed (possibly cyclic) graph where each edge is labelled with a form, and X is a designated root node in the graph. A term matches the shape predicate if its syntax tree can be "bent into shape" to match a subgraph such that each form in the term lies atop a corresponding edge in the graph (edges may be used more than once), and groups of parallel composition, `!`, and `0` lie within a single node in the graph.

The formal structure of Poly★ uses graphs where node names are just opaque identifiers and the meaning is given by edge labels. When *displaying* the graphs (as in Fig. 1) we have found it useful to put each edge label inside the edge's target node. Of course this can't be done in the rare cases when two edges that share a target disagree.

Graphs alone are not enough to guarantee a finite type for every term. For example, the term `<x> | !(y).<y.y>` can (given the reduction rules of MA) evolve into terms with messages that contain arbitrarily long chains of `x`'s *within* a single form. We need to abstract over messages such that an infinity of forms that look alike except having messages of different length within them can be described by the same shape graph label. This is the job of **message types** μ, which are defined in Figure 5.

The message type $\{f_1, \ldots, f_k\}*$ describes any message built from the any of forms f_i — *except* messages that are single names; such a message is matched by the message type $\{x\}$ instead. When $\{x\}$ is the *only* message type that matches x, we can see unambiguously from a message type whether $\bullet$ will result from trying to substitute a message it matches into an element position. We use **element types** ε and **form types** φ to build form-like structures out of message types and non-message elements.

The syntax and semantics of shape predicates is defined in Figure 5. To save space and present the basic theory more clearly we do not handle *name restriction*; how to treat it is described in [17]. We have also omitted a third form of message types, sequenced message types, which allow more precise types and are defined in [17, sec. 5.2].

Define the **meaning** of message/element/form types and of shape predicates by

$$[\![\mu]\!] = \{M \mid \vdash M : \mu\} \quad [\![\varepsilon]\!] = \{E \mid \vdash E : \varepsilon\} \quad [\![\varphi]\!] = \{F \mid \vdash F : \varphi\} \quad [\![\pi]\!] = \{P \mid \vdash P : \pi\}$$

Proposition 3.1. *The meanings of shape predicates respect the structural congruence: If $P \equiv Q$ then $\vdash P : \pi \iff \vdash Q : \pi$ for all π.* □

Let $\mu_1 * \mu_2$ be the least message type whose meaning includes $M.N$ for all $M \in [\![\mu_1]\!], N \in [\![\mu_2]\!]$. With the language of message types presented here (omitting sequenced message types from [17]), $\mu_1 * \mu_2$ always has the form $\{f_1, \ldots, f_k\}*$, where the f_i's are all the sub-forms that appear in either μ_1 or μ_2 in some canonical order (for this purpose the sub-form x is considered to appear in $\mu = \{x\}$). The $\mu_1 * \mu_2$ operation is associative.

3.2 Flow edges and subtyping. The only part of the shape predicate syntax of Figure 5 that has yet not been explained is the **flow edges** $X \mathrel{-\!\!\boxed{\mathcal{T}}\!\!\to} Y$. They are not used at all in the above definition of the *meaning* of the shape graph, but they will be important for distinguishing between types and non-types. In brief, the flow edge $X \mathrel{-\!\!\boxed{\mathcal{T}}\!\!\to} Y$ asserts that there may be a reduction where a process described by X is moved to Y and in the process incurs a substitution described by $\mathcal{T}$.

Alternatively, $X \mathrel{-\!\!\boxed{\mathcal{T}}\!\!\to} Y$ can be viewed as a *demand* that whenever $P \in [\![\langle G \,|\, X\rangle]\!]$ and Q arises by applying a substitution described by $\mathcal{T}$ to P, it must hold that $Q \in [\![\langle G \,|\, Y\rangle]\!]$. Because flow edges do not contribute to the meaning of shape predicates, there is no guarantee that this demand is actually satisfied for a shape predicate that contains the flow edge. This is a global property of the shape graph, and we will shortly define a class of **flow closed** shape graphs where the interpretation of flow edges is always true.

An important special case is when $\mathcal{T} = \varnothing$, where the process moves *without* any substitution. Then $X \mathrel{-\!\!\boxed{\varnothing}\!\!\to} Y$ can also be viewed as an assertion that $\langle G \,|\, X\rangle$ is a *subtype* of $\langle G \,|\, Y\rangle$, or, symbolically, that $[\![\langle G \,|\, X\rangle]\!] \subseteq [\![\langle G \,|\, Y\rangle]\!]$. We therefore also speak of $X \mathrel{-\!\!\boxed{\varnothing}\!\!\to} Y$ as a **subtyping edge**.

Write $\vdash \mathcal{S} : \mathcal{T}$ iff $\mathrm{Dom}\,\mathcal{S} = \mathrm{Dom}\,\mathcal{T}$ and $\vdash \mathcal{S}(x) : \mathcal{T}(x)$ for all $x \in \mathrm{Dom}\,\mathcal{S}$.

Define the action of type substitution on subforms and message/element types by the rules in Figure 6. This definition ensures that $[\![\mathcal{T}^{\mathbf{M}}\mu]\!]$ contains the result of every *term* substitution $\mathcal{S}^{\mathbf{M}}M$ where $\vdash \mathcal{S} : \mathcal{T}$ and $\vdash M : \mu$, and likewise for elements.

Definition 3.2. *The shape graph G is **flow closed** iff whenever G contains $X \xrightarrow{\varphi} Y$ and $X \mathrel{-\!\!\boxed{\mathcal{T}}\!\!\to} Z$ such that $\mathrm{BN}(\varphi) \cap \mathrm{Dom}\,\mathcal{T} = \varnothing$, then there is a W such that G contains $Y \mathrel{-\!\!\boxed{\mathcal{T}}\!\!\to} W$ and additionally it holds that*

$$\mathcal{T}^{\mathbf{f}}(x_0 \ldots x_k) = \begin{cases} \mathcal{T}(x_0) & \text{when } k = 0 \text{ and } x_0 \in \operatorname{Dom} \mathcal{T} \\ \{\mathcal{T}^{\mathbf{E}} x_0 \ldots \mathcal{T}^{\mathbf{E}} x_k\}* & \text{otherwise} \end{cases}$$

$$\mathcal{T}^{\mathbf{M}}\{f_1, \ldots, f_k\}* = \{\}*_*\mathcal{T}^{\mathbf{f}} f_{1*}\cdots_*\mathcal{T}^{\mathbf{f}} f_k \qquad \mathcal{T}^{\mathbf{M}}\{x\} = \begin{cases} \mathcal{T}(x) & \text{if } x \in \operatorname{Dom} \mathcal{T} \\ \{x\} & \text{otherwise} \end{cases}$$

$$\mathcal{T}^{\mathbf{E}} x = \begin{cases} y & \text{when } \mathcal{T}^{\mathbf{M}}\{x\} = \{y\} \text{ for some } y \\ \bullet & \text{otherwise} \end{cases}$$

$$\mathcal{T}^{\mathbf{E}} \langle \mu_1, \ldots, \mu_k \rangle = \langle \mathcal{T}^{\mathbf{M}} \mu_1, \ldots, \mathcal{T}^{\mathbf{M}} \mu_k \rangle$$

$$\mathcal{T}^{\mathbf{E}} (x_1, \ldots, x_k) = (x_1, \ldots, x_k)$$

Fig. 6. *The action of type substitutions. $\mathcal{T}^{\mathbf{f}} f$ is the action on subforms, $\mathcal{T}^{\mathbf{M}} \mu$ is the action on message types, and $\mathcal{T}^{\mathbf{E}} \varepsilon$ is the action on element types.*

Let a **type instantiation** $\mathcal{U}$ map $\boxed{\boxed{\mathring{x}}}$ to $\boxed{X} \setminus \{\bullet\}$, $\boxed{\boxed{\mathring{m}}}$ to $\boxed{\mu}$, and $\boxed{\boxed{\mathring{p}}}$ to $\boxed{X}$. It applies to element and form templates completely componentwise (giving element and form types), just like term instantiations do. The relation betwen type instantiations and process templates are given by this inference system:

$$\frac{X = \mathcal{U}(\mathring{p})}{\mathcal{U} \models_{\mathsf{L}} \mathring{p} : \langle G \mid X \rangle} \qquad \frac{(\mathcal{U}(\mathring{p}) \multimap \boxed{\varnothing} \rightarrow X) \in G}{\mathcal{U} \models_{\mathsf{R}} \mathring{p} : \langle G \mid X \rangle} \qquad \frac{(\mathcal{U}(\mathring{p}) \multimap \boxed{\ldots, \mathcal{U}(\mathring{x}_i) \mapsto \mathcal{U}^{\mathbf{S}} \underline{S}_i, \ldots} \rightarrow X) \in G}{\mathcal{U} \models_{\mathsf{R}} \{\ldots, \mathring{x}_i := \underline{S}_i, \ldots\}\, \mathring{p} : \langle G \mid X \rangle}$$

$$\frac{}{\mathcal{U} \models_s \mathsf{0} : \pi} \qquad \frac{\mathcal{U} \models_s \underline{P}_1 : \pi \qquad \mathcal{U} \models_s \underline{P}_2 : \pi}{\mathcal{U} \models_s \underline{P}_1 \mid \underline{P}_2 : \pi} \qquad \frac{(X \xrightarrow{\mathcal{U}^{\mathbf{F}} \underline{F}} Y) \in G \qquad \mathcal{U} \models_s \underline{P} : \langle G \mid Y \rangle}{\mathcal{U} \models_s \underline{F}.\underline{P} : \langle G \mid X \rangle}$$

where $\mathcal{U}^{\mathbf{S}} \mathring{x} = \{\mathcal{U}(\mathring{x})\}$ and $\mathcal{U}^{\mathbf{S}} \mathring{m} = \mathcal{U}(\mathring{m})$. The rules for template processes have an L variant and an R variant; the variable letter s ranges over L and R.

As a special exception, $\mathcal{U} \models_s \underline{P} : \pi$ is *not* considered to hold if $\mathcal{U}(\mathring{x}_1) = \mathcal{U}(\mathring{x}_2)$ for $\mathring{x}_1 \neq \mathring{x}_2$ such that $\mathring{x}_1$ occurs in $\underline{P}$ below a form template containing a binding element $(\ldots, \mathring{x}_2, \ldots)$.

Fig. 7. *Matching of reduction rules to shape graphs.*

1. *If $\varphi = x$ and $\mathcal{T}(x) = \{y\}$, then G contains $Z \xrightarrow{y} W$.*
2. *If $\varphi = x$ and $\mathcal{T}(x) = \{f_1, \ldots, f_k\}*$, then $W = Z$ and G contains $Z \xrightarrow{f_i} Z$ for $1 \leq i \leq k$.*
3. *If none of the above apply and $\varphi = \varepsilon_0 \ldots \varepsilon_k$, then G contains $Z \xrightarrow{\mathcal{T}^{\mathbf{E}} \varepsilon_0 \ldots \mathcal{T}^{\mathbf{E}} \varepsilon_k} W$.*

We call a shape predicate $\langle G \mid X \rangle$ flow closed iff its G component is. □

Intuitively, a flow-closed graph is one where the intuitive meanings of flow edges are true. That is the content of the following theorem:

Proposition 3.3. *Let G be flow closed and contain $X \multimap \boxed{\mathcal{T}} \rightarrow Y$. Assume that $\vdash \mathcal{S} : \mathcal{T}$ and that $\mathrm{BN}(P) \cap \operatorname{Dom} \mathcal{T} = \varnothing$. Then $\vdash P : \langle G \mid X \rangle$ implies $\vdash \mathcal{S}^{\mathbf{P}} P : \langle G \mid Y \rangle$* □

The assumption that $\mathrm{BN}(P) \cap \operatorname{Dom} \mathcal{T} = \varnothing$ will be true in our uses because we are assuming that all terms are well-scoped.

3.3 Semantic and syntactic closure; types. Call the shape predicate π **semantically closed** with respect to $\mathcal{R}$ iff $\vdash P : \pi$ and $\mathcal{R} \vdash P \hookrightarrow Q$ imply $\vdash Q : \pi$.

As described above, we want *types* to be semantically closed shape predicates. But it is not easy to recognise semantic closure.[6] We will therefore define a restricted, but easier to decide, class of *syntactically* closed shape predicates, which will be the types.

Figure 7 defines a way to match process templates directly to type graphs without going via the process semantics from Fig. 4. A type instantiation applies to message, element, and form templates much like the $\mathcal{V}$'s in Fig. 4, but the process part is different because a type instantiation maps process metavariables to nodes in the shape graph rather than processes. This is best illustrated with an example. Assume we wish to find out whether $\langle G_0 \,|\, \mathtt{R} \rangle$ is closed with respect to $\mathcal{R}_{\text{eat}}$, where

$$G_0 = \left\{ \begin{array}{ccccccc} \mathtt{R} & \xrightarrow{\mathtt{t[]}} & \mathtt{X} & \xrightarrow{\mathtt{p[]}} & \mathtt{Y} & \xrightarrow{\mathtt{eat\ q}} & \mathtt{Z} \xrightarrow{\mathtt{foo}} \mathtt{T} \\ & & & \searrow^{\mathtt{q[]}} & & \searrow^{\mathtt{q[]}} & \\ & & & & \mathtt{W} & & \mathtt{V} \xrightarrow{\mathtt{bar}} \mathtt{U} \\ & & & & & \searrow^{\mathtt{s[]}} & \\ & & & & & & \mathtt{S} \end{array} \right\}$$

$\mathcal{R}_{\text{eat}}$ has only one **reduce** rule, and we look for matches of its left-hand side, that is $(X_0, \mathcal{U}_0)$ such that X_0 can be reached from the root by edges with labels of the shape $x\mathtt{[]}$ and $\mathcal{U}_0 \models_{\mathtt{L}} \mathtt{a[eat\,b \mid P] \mid b[Q]} : \langle G_0 \,|\, X_0 \rangle$. The only such pair is

$$X_0 = \mathtt{X}, \quad \mathcal{U}_0 = \{\mathtt{a} \mapsto \mathtt{p}, \mathtt{b} \mapsto \mathtt{q}, \mathtt{P} \mapsto \mathtt{Y}, \mathtt{Q} \mapsto \mathtt{W}\}$$

For the graph to be syntactically closed, the same X and $\mathcal{U}_0$ must match the *right*-hand side of the rule, i.e. $\mathcal{U}_0 \models_{\mathtt{R}} \mathtt{a[P \mid b[Q]]} : \langle G_0 \,|\, \mathtt{X} \rangle$. This turns out not to be true; if one tries to construct at derivation it turns out that the judgement $\mathcal{U}_0 \models_{\mathtt{R}} \mathtt{Q} : \langle G_0 \,|\, \mathtt{V} \rangle$ is needed, and that can only be true if $(\mathcal{U}_0(\mathtt{Q}) = \mathtt{W} \multimap \mathtt{V}) \in G_0$ which is not the case.

Thus the graph is not syntactically closed for $\mathcal{R}_{\text{eat}}$, and indeed it is not semantically closed either, because $P_1 = \mathtt{t[p[eat\ q] \mid q[s[0]]]}$ is in $[\![\langle G_0 \,|\, \mathtt{R} \rangle]\!]$ but $P_2 = \mathtt{t[p[q[s[0]]]]}$ is not, even though $\mathcal{R}_{\text{eat}} \vdash P_1 \hookrightarrow P_2$. If we add the missing edge, giving $G_1 = G_0 \cup \{\mathtt{W} \multimap \mathtt{V}\}$, we do get $\mathcal{U}_0 \models_{\mathtt{R}} \mathtt{a[P \mid b[Q]]} : \langle G_1 \,|\, \mathtt{X} \rangle$, but G_1 is not *flow closed*, and we need to add even more edges to make it so. In the end we get

$$G_2 = \left\{ \begin{array}{ccccccc} \mathtt{R} & \xrightarrow{\mathtt{t[]}} & \mathtt{X} & \xrightarrow{\mathtt{p[]}} & \mathtt{Y} & \xrightarrow{\mathtt{eat\ q}} & \mathtt{Z} \xrightarrow{\mathtt{foo}} \mathtt{T} \\ & & & \searrow^{\mathtt{q[]}} & & \searrow^{\mathtt{q[]}} & \\ & & & & \mathtt{W} \dashrightarrow \mathtt{V} & & \xrightarrow{\mathtt{bar}} \mathtt{U} \\ & & & & & \searrow^{\mathtt{s[]}} & \searrow^{\mathtt{s[]}} \\ & & & & & & \mathtt{S} \dashrightarrow \mathtt{VV} \end{array} \right\}$$

where, as in Fig. 1, subtyping edges are shown as dashed lines. The shape predicate $\langle G_2 \,|\, \mathtt{R} \rangle$ is syntactically and semantically closed.

We are now ready do define syntactic closure formally:

Definition 3.4. *Let the shape predicate $\pi = \langle G \,|\, X \rangle$ be given. The set of **active** nodes for $\mathcal{R}$, written* $\mathsf{active}_{\mathcal{R}}(\pi)$*, is the least set A of nodes which contains X and such that for all $Y \in A$ and all* **active**$\{\mathring{p}$ **in** $\underline{P}\} \in \mathcal{R}$*, it holds that $\mathcal{U} \models_{\mathtt{L}} \underline{P} : \langle G \,|\, Y \rangle$ implies $\mathcal{U}(\mathring{p}) \in A$.* □

[6] E.g., let $G = \{\mathtt{Y1} \xleftarrow{\mathtt{c}} \mathtt{Y0} \underset{\mathtt{b}}{\overset{\mathtt{b}}{\rightleftarrows}} \mathtt{X0} \xrightarrow{\mathtt{(a)<\{b\}*>}} \mathtt{X1} \xrightarrow{\mathtt{a}} \mathtt{X2} \xrightarrow{\mathtt{b}} \mathtt{X3} \xrightarrow{\mathtt{a}} \mathtt{X4} \xrightarrow{\mathtt{c}} \mathtt{X5}\}$. Then $\langle G \,|\, \mathtt{X0} \rangle$ happens to be semantically closed with respect to $\{\mathbf{reduce}\{\mathtt{(a)<M>.P} \hookrightarrow \{\mathtt{a} := \mathtt{M}\}\mathtt{P}\}\}$, but it is not trivial to see this in a systematic way.

Definition 3.5. *G is **locally closed** at X with respect to $\mathcal{R}$ iff whenever $\mathcal{R}$ contains* **reduce**$\{\underline{P}_1 \hookrightarrow \underline{P}_2\}$ *it holds that $\mathcal{U} \models_{\mathsf{L}} \underline{P}_1 : \langle G \,|\, X \rangle$ implies $\mathcal{U} \models_{\mathsf{R}} \underline{P}_2 : \langle G \,|\, X \rangle$.* □

Definition 3.6. *The shape predicate $\pi = \langle G \,|\, X \rangle$ is **syntactically closed** with respect to $\mathcal{R}$ iff G is flow closed and also locally closed at every $X \in \mathsf{active}_{\mathcal{R}}(\pi)$. When this holds, we call π an ($\mathcal{R}$-)**type**.* □

Checking that a purported type is really syntactically closed is algorithmically easy; see [17] for details.

Theorem 3.7 (Subject reduction). *If π is syntactically closed with respect to $\mathcal{R}$, then it is also semantically closed with respect to $\mathcal{R}$.* □

3.4 What to do with types. Once we have a type, what can we use it for? An obvious possibility is to check whether the term may "go wrong". The user (or, more likely, the designer of a programming environment that uses Poly$\star$) specifies what "going wrong" means. It is a clear indication of error if $\bullet$ turns up in an active position, but opinions may differ about if a $\bullet$ is produced at a place in process tree that never becomes active.

One can also imagine very application-specific properties to check for, for example "this process can never evolve to a configuration where an ambient named a is inside one named b". This is easy to check for in the shape graph. Alternatively, one may want to write this as a rule, to have Poly$\star$ do the checking: **reduce**$\{\mathsf{b}[\mathsf{a}[\mathtt{P}] \mid \mathtt{Q}] \hookrightarrow \bullet.0\}$. The ability to write such rules is one of the reasons why Meta$\star$ does not distinguish strictly between "names" and "keywords".

Poly$\star$ makes it fairly easy to check *safety properties* like "no unauthorised ambients (e.g., a) inside secure ambients (e.g., b)", but there are also questions of safety that Poly$\star$ cannot help determine. This includes properties that depend on the *order* in which things happen, such as the "correspondence assertions" often used to specify properties of communication protocols. There are type systems for process calculi that can reason about such temporal properties (for example, [13] for the π-calculus), but we are aware of none that also handle locations and ambient-style mobility.

4 Type inference for Poly$\star$

Assume now that we are given a process term P and a ruleset $\mathcal{R}$; we want to produce an $\mathcal{R}$-type for P. It is trivial to construct *some* type for P – one with a single node and a lot of edges in the shape graph. However, such a type may need to contain $\bullet$'s and thus not prove that P "cannot go wrong". In this section we discuss how to automatically infer more informative types.

We do not know how to do type inference that is complete for the full Poly$\star$ type system; it allows too many types. Therefore we begin by defining a set of *restricted* types, for which we *can* have complete type inference.

Definition 4.1. *Write $\varphi_1 \approx \varphi_2$ iff $[\![\varphi_1]\!] \cap [\![\varphi_2]\!] \neq \varnothing$.* □

The $\approx$ relation is close to being equality. The only way for two non-identical φ's to be related by $\approx$ is if they contain message types of the shape $\{\cdots\}\star$. It is relatively safe to imagine the $\approx$ is just a fancy way to write $=$, at least to a first approximation.

Definition 4.2. *G satisfies the* ***width restriction*** *iff whenever it contains* $X \xrightarrow{\varphi} Y$ *and* $X \xrightarrow{\varphi'} Y'$ *with* $\varphi \approx \varphi'$*, it holds that* $Y = Y'$. □

Definition 4.3. *G satisfies the* ***depth restriction*** *iff whenever it contains a chain* $X_0 \xrightarrow{\varphi_1} X_1 \xrightarrow{\varphi_2} \cdots \xrightarrow{\varphi_k} X_k$ *with* $\varphi_1 \approx \varphi_k$*, it holds that* $X_1 = X_k$. □

Our type inference algorithm only produces types that satisfy both restrictions. The width restriction means that when type inference needs to add an outgoing edge from a node in the graph, it never has to choose between reusing an existing edge starting there and creating a new edge with a fresh target node, because the latter is forbidden by the restriction when there is any reusable edge. The depth restriction bounds the length of a simple path in a shape graph that can be constructed with a given set of names and a given maximal form arity, and so also bounds the total number of nodes in the graph. Therefore the closing operation described below cannot keep adding edges and nodes to the graph indefinitely and will eventually stop. (These two restrictions replace the notions of "discrete" and "modest" types in [2], which sometimes admitted slightly more precise types, but were very complex and hard to understand).

In [17] we describe a feature of our implementation which allows it to loosen the two restrictions by tracking the origin of each φ in the type graph.

The type inference proceeds in two phases. First we construct a minimal shape predicate which the term matches. Then we **close** the shape predicate — that is, rewrite its shape graph as necessary to make it syntactically closed.

The initial phase is simple. Because shape predicates "look like terms", we can just convert the abstract syntax tree of the term to a tree-shaped shape graph. This graph may or may not satisfy the width and depth restrictions. If it does not, unify the nodes that must be equal.[7] That may cause further violations of the two restrictions; continue unifying nodes as necessary until the restrictions are satisfied.

The closing of the shape graph is where the real work of type inference happens. It happens in a series of steps. In each step, check whether the shape graph is syntactically closed. If it is, the algorithm ends. Otherwise, the lack of closure can only be because edges already in the graph imply that some other edges *ought* to exist (by Definitions 3.2 or 3.5) but do not. In that case, add the new nodes and edges required by the failing rule, and do another round of unifications to enforce the width and depth restrictions.

The width and depth restriction together guarantee that the closure phase terminates. We do not have any good worst-case complexity bounds for the closure phase; instead our implementation allows further restrictions on types to be applied in order to quench blow-ups one might observe with particular calculi and example terms. The tightest restrictions will enforce polynomial complexity, at the cost of losing the possibility of spatial polymorphism. Thus restricted, Poly$\star$ has a strength roughly comparable to current non-polymorphic type systems for ambient-derived calculi.

Theorem 4.4 (Principal typings). *A result of type inference* π *is a principal typing [26] for the input term P: For every* π' *such that* $\vdash P : \pi'$ *it holds that* $[\![\pi']\!] \supseteq [\![\pi]\!]$. □

[7] This is the only way to reach a graph that satisfies the restrictions. If the width and depth restrictions had used $=$ instead of $\approx$, there might also have been the option of rewriting a φ to something larger but different, but there would not be a unique "best way" of doing that.

4.1 Implementation. We have implemented our type inference algorithm. Our implementation is available at the URL ⟨`http://www.macs.hw.ac.uk/DART/software/PolyStar/`⟩, as both a source download and an interactive web interface.

Beyond the features in this paper, our implementation allows fine-tuning of the analysis precision, which influences inference speed as well as inferred type size.

5 Conclusion

Poly✶ extends basic properties of our previous system PolyA to a more general setting:

1. Poly✶ has *subject reduction*. Also, given a process term P and a shape predicate π, one can decide by checking purely local conditions whether π is a *type*, and it is similarly decidable whether P *matches* π. Thus, it is decidable whether a process belongs to a specific type.
2. Poly✶ supports a notion of *spatial polymorphism* that achieves what Cardelli and Wegner [9] 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".
3. The types of Poly✶ are sufficiently precise that many interesting *safety/security properties* can be checked, especially those that can be formulated as questions on the possible configurations that can arise at run-time.

In addition, this paper makes these completely novel contributions:

4. Meta✶ is a syntactic framework that can be instantiated into a large family of mobile process calculi by supplying reduction rules.
5. The *generic type system* Poly✶ works for any instantiation of Meta✶. We have checked that it works for π-calculus, a large number of ambient calculi, and a version of the Seal calculus. In [2] we claimed PolyA would be easy to extend to ambient-like calculi by hand, but extending the proofs for PolyA manually would be tedious. With Meta✶ we have developed the theory to do such extensions fully automatically.
6. For the subsystem of Poly✶ satisfying the *width* and *depth* restrictions, there is a type inference algorithm (which we have implemented) that always successfully infers a *principal type* for any process term. This means that Poly✶ has the potential for *compositional analysis*.
7. The width and depth restriction are more natural and intuitive than the "discreteness" and "modesty" properties with respect to which we showed existence of principal types for PolyA.
8. Poly✶'s handling of *communication* and *substitution* has been redesigned to be more direct and intuitive than in PolyA.

5.1 Related work. Another generic type system for process calculi was constructed by Igarashi and Kobayashi [14]. Like the shape predicates in Poly✶, their types look like process terms and stand for sets of structurally similar processes. Beyond that, however, their focus is different from ours. Their system is specific to the π-calculus and does not handle locations or ambient-style mobility. On the other hand, it is considerably

more flexible than Poly✶ within its domain and can be instantiated to do such things as deadlock and race detection which are beyond the capabilities of Poly✶.

Yoshida [27] used graph types much like our shape predicates to reason about the order of messages exchanged on each channel in the π-calculus. Since this type system reasoned about *time* rather than *location*, it is not directly comparable to Poly✶, despite the rather similar type structure.

The spatial analysis of Nielson et al. [19] produces results that somewhat resemble our shape graphs, but does not have spatial polymorphism.

References

[1] M. Abadi, A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Inform. & Comput.*, 148(1), 1999.

[2] T. Amtoft, H. Makholm, J. B. Wells. PolyA: True type polymorphism for Mobile Ambients. In *IFIP TC1 3rd Int'l Conf. Theoret. Comput. Sci. (TCS '04)*. Kluwer Academic Publishers, 2004.

[3] T. Amtoft, F. Turbak. Faithful translations between polyvariant flows and polymorphic types. In *Programming Languages & Systems, 9th European Symp. Programming*, vol. 1782 of *LNCS*. Springer-Verlag, 2000.

[4] G. Boudol. The π-calculus in direct style. In *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, 1997.

[5] 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.

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

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

[10] G. Castagna, G. Ghelli, F. Z. Nardelli. Typing mobility in the Seal calculus. In K. G. Larsen, M. Nielsen, eds., *CONCUR*, vol. 2154 of *LNCS*. Springer-Verlag, 2001.

[11] S. Chaki, S. K. Rajamani, J. Rehof. Types as models: Model checking message-passing programs. In *Conf. Rec. POPL '02: 29th ACM Symp. Princ. of Prog. Langs.*, 2002.

[12] 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.

[13] A. D. Gordon, A. S. A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoret. Comput. Sci.*, 300(1–3), 2003.

[14] A. Igarashi, N. Kobayashi. A generic type system for the pi-calculus. In *Conf. Rec. POPL '01: 28th ACM Symp. Princ. of Prog. Langs.*, 2001.

[15] F. Levi, C. Bodei. A control flow analysis for safe and boxed ambients. In *Programming Languages & Systems, 13th European Symp. Programming*, vol. 2986 of *LNCS*. Springer-Verlag, 2004.

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

[17] H. Makholm, J. B. Wells. Instant polymorphic type systems for mobile process calculi: Just add reduction rules and close. Technical Report HW-MACS-TR-0022, Heriot-Watt Univ., School of Math. & Comput. Sci., 2004.

[18] R. Milner, J. Parrow, D. Walker. A calculus of mobile processes. *Inform. & Comput.*, 100(1), 1992.

[19] H. R. Nielson, F. Nielson, H. Pilegaard. Spatial analysis of BioAmbients. In R. Giacobazzi, ed., *Static Analysis: 11th Int'l Symp.*, vol. 3148 of *LNCS*, Verona, Italy, 2004. Springer-Verlag.

[20] J. Palsberg, C. Pavlopoulou. From polyvariant flow information to intersection and union types. *J. Funct. Programming*, 11(3), 2001.

[21] S. M. Pericas-Geertsen. *XML-Fluent Mobile Ambients*. PhD thesis, Boston University, 2001.

[22] I. Phillips, M. G. Vigliotti. On reduction semantics for the push and pull ambient calculus. In *Theoretical Computer Science: 2nd IFIP Int'l Conf.*, vol. 223 of *IFIP Conference Proceedings*. Kluwer, 2002.

[23] G. Sander. Graph layout through the VCG tool. In R. Tamassia, I. G. Tollis, eds., *Graph Drawing: DIMACS International Workshop, GD '94*, vol. 894 of *LNCS*. Springer-Verlag, 1994.

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

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

[26] 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.

[27] N. Yoshida. Graph types for monadic mobile processes. In *Foundations of Software Technology and Theoret. Comput. Sci., 16th Conf.*, vol. 1180 of *LNCS*. Springer-Verlag, 1996.